

Final project report: Bitcoin transactions

Programming language: Java

Graph database: Neo4j

Query language: Cypher

*Peer To Peer Systems and Blockchains*

*Ornela Danushi*

Design choices:

The first decision that I took into account was regarding the database where to store all the Bitcoin transactions.

I opted for a type of database that belongs to the graph database family because they have some useful pros that meet the project’s needs too. They are:

* there is a big amount of data that can be handled in an efficient way thanks to proper optimizations that this platform offers;
* the data set contains some erroneous elements that can be discovered in each moment, also after have set or checked some properties on them;
* either the relations between entities or the nodes themselves of compositions, can be always modified with affordable penalty in performance;
* the overall composing structure can be visually useful to understand and deduce something.

The second decision was that, referring to the language with which I could interact with the database, I opted for Java because the chosen graph database already offers a lot of predefined primitives that allow the connection between the server with the database and the Java program with the queries. For this purpose is set an instance of a driver that can be used as a tool with which we activate the sessions with the database. If more than one session is present inside a database transaction means that they can be handled simultaneously in parallel. But, of course, we can have this comfortable situation only if the sessions are independent from each other.

Moreover, this connection passes through a protocol, like http but more oriented to handle these types of requests, called bolt. By using it, there is the only need to specify the url of the server, in which the database is located, and the password to get access to it.

As for instance:

*static Driver driver = GraphDatabase.driver("bolt://localhost:7687",*

*AuthTokens.basic("neo4j", "grafico")); //userID and password*

Start the overall structure

Every time we run the program, we erase already present data so we always consider a new clean scenario.

Then we get the data from 3 different csv files. The reading of these files can happen in parallel, since they have independent information. This operation builds the entities of our database, that for simplicity and unanimous conformity we call nodes. Each element in a graph database is characterized by some properties, first of all an ID and then followed by the others needed.

I have to admit that I introduced some further fields to the nodes, which were not present in the corresponding csv files, such as:

* a boolean property called *marked*, initialized to false but as soon as we check if the transaction is valid then will eventually change its value, in the case it will not then the transaction it refers to will be deleted together with some remaining isolated nodes;
* an integer value *in*, initialized to 0 and will contain the sum of all the bitcoin values achieved in input. To it corresponds a list called *input\_list*, initialized to empty and will contain the outputIDs of the values already counted in the *in* summation;
* analogously, an integer value *out*, initialized to 0 and will contain the sum of all the bitcoin values given in output. To it corresponds a list called *output\_list*, initialized to empty and will contain the outputIDs of the values already counted in the *out* summation.

All these new introduced fields will be useful for the evaluation of validity of the nodes, which will be more clear later.

Once we have finished building the nodes, we scan them and in case of matching properties we link them. I have called the one relationship from Input to Transaction *appeared\_in*, the one from Output to Transaction *appeared\_from*, and the one referring both to that linking the ID of the Output with the output\_id of the Input and both that of pk\_id with the sig\_id calling it *used\_with*.

Till now this creation of the database corresponds to the one specified in the figure.

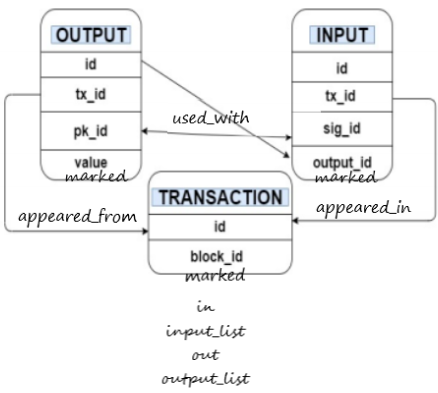


Fig.1: Format of CSV files together with new fields.

Analysis

1. Discover erroneous data

The analysis phase cannot start if we don’t first check if our data are all valid.

The first fact taken into account, for the purpose of discovering erroneous elements, was regarding the balance of each transaction. It means that all the bitcoin values are counted both in input and both in output toward a transaction. To avoid mistakes in counting more than once, a list is used in which are saved the already counted outputIDs. Then are compared the *in* value with the *out* once and if those in input are smaller than those in output then there is a mistake since a transaction cannot spend more than how much it owns. I suppose the eventual fees are already inside the indicated amount and so those elements have to be deleted definitely. Of course, this is not the case of the transactions arising from the award nodes in which the *in* value is 0 and the *out* value is greater.

To decide about validity of the data I have also to consider if the conditions imposed by the Bitcoin protocol are satisfied.

Among them I have checked if the transactions are inside a chain that is at least as long as the 6 confirmation rule may do. It means, if there is a path long enough from a transaction which is linked to others that have to belong to different blocks.

If this condition is satisfied, the *marked* field introduced before is signed to true otherwise it stays false. Then we scan all the nodes and delete all transactions that have this field remaining as false. By this deletion, some nodes could remain unlinked, so if there are isolated Input and Output nodes, they are also deleted.

Another consideration done, to check the validity of the data, was the double spending, it means that, if the same transaction belongs to different blocks in which are followed different chains. To discover this, I grouped each transaction to its belonging block. This was achieved by saving all the blockIDs of each transaction in a Java set. I opted for this collection because it does not admit duplicates. Then each blockID becomes a node in the database, by which the transactions belonging to it were linked. It was at this moment that, by counting all the different transactionIDs and all the different blockIDs I soon discovered there were more Transaction nodes than Block nodes and so there were more transactions linked to a single block. But by counting how many the same transaction appears in different blocks, the result was 0 so there was no double spending.

For now the database is something like is shown in the following figure.

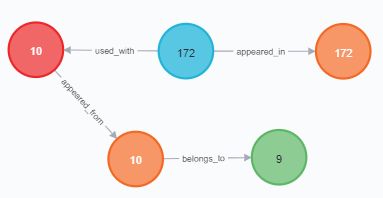


Fig.2: In blue the Input nodes, in red the Output nodes, in green the Block nodes and in orange the Transaction nodes. It has to be read as: transaction 10, belonging to block 9, has generated the values contained in output 10, which are used as input toward the transaction 172.

1. Compute UTXOs

At this moment we are ready to start with some interesting considerations about the analysis.

For instance, we compute the max amount of present UTXOs for each address, with the aim of discovering where the stationary points of bitcoins are in our money flow. To see this, we take into consideration all Output and Transaction nodes. If there is not a path linked to the Output node of type *used\_with*, it means that the values of it are not given as input to another transaction so are not spent in another transaction. We sum all those values per address and give the max one. We save locally in the Java program the address and the corresponding max UTXO value in a hashtable, since it will be useful for later clustering reasons.

As an example of the result we can see the following figure.

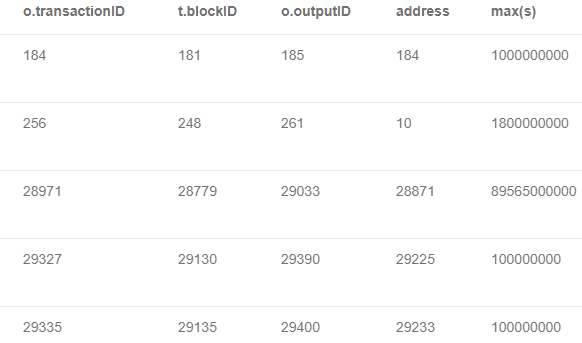


Fig.3: 5 examples of the resulting tuple composed by <transactionID, blockID, outputID, pkID, max(sum(value))>

And to the question “*Which UTXO (TxId, blockId, output index and address) has the highest associated value?*” I answer with “*The highest associated UTXO value of: 3000000000000 belongs to transactionID 136084 of the blockID 88148 with outputID 165004 and address 174680*”. In the later elaboration, this address is linked to a user who is identified by also other address, he/she has

“*Lowest address: 134505*

*Tot UTXOs owned: 3000000000000 of which the max UTXO owned: 3000000000000*”

To that entity I also discovered that the “*TransactionID sending the greatest amount of bitcoins: 139898*”.

1. Compute cluster entities

Now I proceeded with the clustering of the data with the aim of discovering the entities which have the controls of the addresses. Before applying the requested heuristics I thought it was useful to highlight another one not considered as a cluster of nodes.

I opted for linking together all the reward inputs, so those inputs which outputID was -1 and the sigID was 0. Another property of those inputs was the fact that the only relationships they had with other nodes was of the type *appeared\_in* toward a Transaction node.

This was done because also the system, or who else, that can give the rewards is someone who has a sort of control in the bitcoin scenario.

We can see how this can appear graphically in the database, with the following figure.

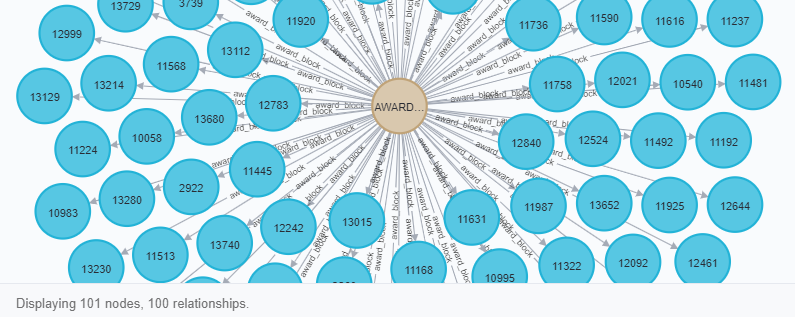


Fig.4: In brown the Award\_Cluster node together with some Inputs nodes with which there are the *award\_block* relations.

After this, I posed my attention to what concerns the first asked heuristic; the linking of all the common input addresses. To realize it I was searching for the Transaction, Input and Output nodes which had the same matching transactionID field, once I found them I linked together the Input with the Output node through the *same\_user* relationship. This was justified by the fact that because they refer to the same Transaction activity, then the input of bitcoins are addressed to one or several addresses of the Output node owned by the same entity that Transaction refers to. So a transaction can address bitcoins to one or more addresses that belong to the same user. Once believed to this then it is easy to understand that the serial control heuristic is just a simple case of this one, and for this reason it is already satisfied. As a matter of fact it refers to the base case of a single input toward a transaction that generates a single output. This serial control heuristic, as shown also in the figure, refers also to the rewarding nodes.

By doing this the database is something looking like as shown.

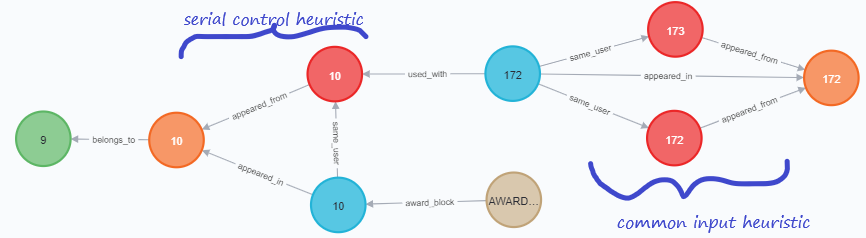


Fig.5: Application of heuristics. The graph has to be read as: from the Award\_Block arrives some bitcoins given as input 10 toward the transaction 10 which will generate the output 10. For the application of the serial control heuristic the public address of the output 10 is the one owned by the input 10. Then bitcoins of output 10 flow with input 172 toward the transaction 172, and some part goes into the output 172 and the remaining part to the output 173. For the application of the common input heuristic, these last 2 outputs are of the same user of the input 172, so he/she knows the secret keys of them.

A cluster here appears graphically as a set of edges between Input and Output nodes of type *same\_user*.

The consideration about transitivity, the one referring to the fact that 2 clusters are merged if they contain at least 2 addresses which appear together as input in at least one transaction, is difficult to be realized also by building an algorithm for this specific purpose. We could go through a confusionary and not useful story. So, I opted to limit the realization of considerations only graphically and to work inside the Java program later for other purposes, for instance by studying individually the addresses and the secret keys, with the adoption of some considerations that can bring to deduce something more about the eventual merging of the clusters.

To convince ourselves about the already satisfied transitivity property we can see the following example.

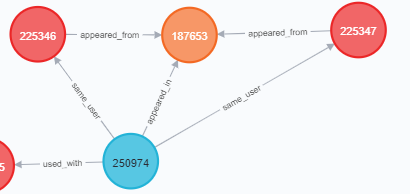


Fig.6: UserX of the input 250974 has the public addresses of the output 225346 and 225347

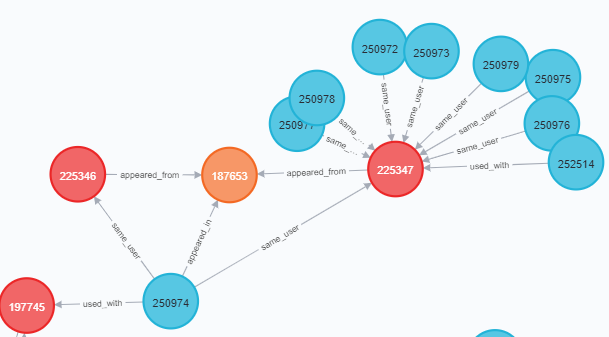


Fig.7: UserY of all the shown inputs has the public address of the output 225347

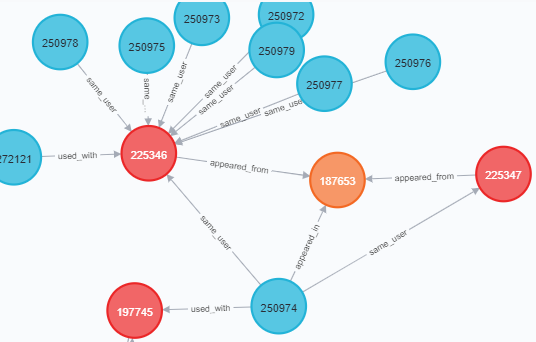


Fig.8: UserZ of all the shown inputs has the public address of the output 225346

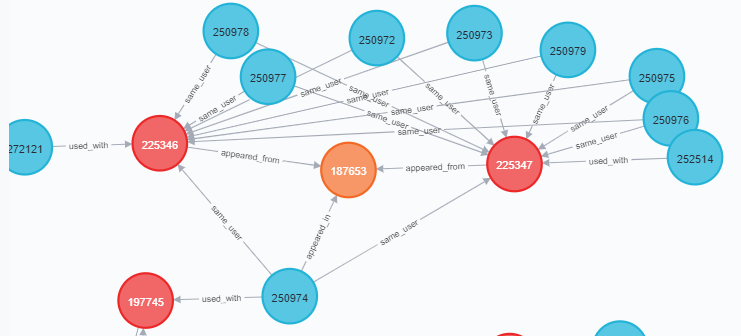


Fig.9: UserZ has the same public address of the output 225346 which is also the same of the output 22347 of UserY, so for transitivity UserZ = UserY. But UserX of the input 250974 has the public addresses of the output 225346 and 225347 so also UserX = UserZ = UserY. The transitivity is demonstrated.

In the clustered transaction graph I found the “*Length of the longest payment path: 378735*”.

The proposed clustering methods are not accurate and to prove this detection I have used some hashtables to hold the needed data such as one associating each transaction to the addresses it refers to and another one associating the addresses to the passwords that I could use to get access to them.

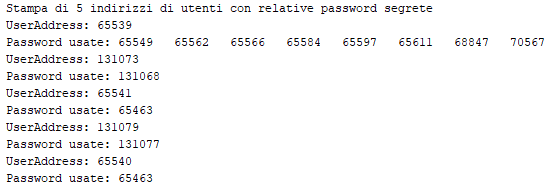


Fig.10: To some public addresses I can access with several secret passwords.

First of all, both the heuristics yield a useful clustering of users, but restricting ourselves to only these heuristics does not tell the whole story. They limit the cluster identification to only those outputs which have inputs in the same transaction, so a transaction is used as the user identification. While a user can have more inputs, in which are specified the used secret keys, and more outputs, in which are specified the public keys, or the addresses, of that user. But if the same user is also the one exploiting another transaction, this is impossible to link with him/her, and also the transitivity property does not help so much.

I thought that maybe we can associate a same user, or entity, by identifying him/her with the same secret keys that he/she uses for different addresses, but theoretically this is not so trustable since different users could use the same secret key. Instead, if we could think that this is unique, maybe a hash built by taking into account also some info which are unique per user, then we could link together as the same user by using the same secret key in different addresses of different transactions. For this reason we can get false negatives as shown in the program. In conclusion to this observation, I could change the proposed heuristics by adding also this explained constraint on the secret keys.

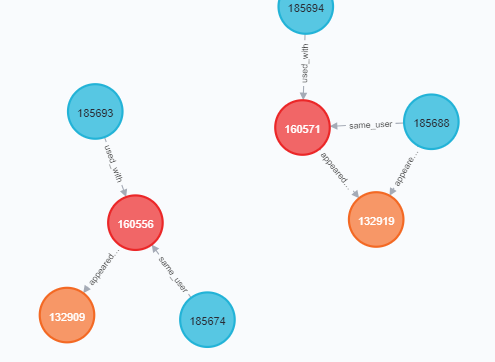


Fig.11: A false negative between these public addresses: 131073 (**outputID:**160556 ) and 131084 (**outputID:**160571). They refer to different transactions: 132909 and 132919 but are accessed with the same secret key (**sigID:**131068). So they have to be of the same user while instead there does not appear the *same\_user* relationships among them.

As false positives we could consider all those cases in which a transaction is linked to several outputs. We say that the output is of the same user of the inputs that the transaction has. But, this is not so clear to be for sure. It may not always be a legal deduction. It could happen in cases either in which an input is sent to different addresses of the same user with a single transaction, or it could happen in cases in which an input is sent to different addresses of different users with a single transaction. With the applied heuristics we always make the first assumption, while this may be not always true. I don't show the code for this case since it is just a semantic consideration.

For all these considerations, our heuristics lack robustness in the face of network changeability and lack of enough associable information with the entities.

Moreover, while false negatives are quite affordable with more restrictions, false positives conduct us in a worst scenario. Because, falsely linking even a small number of addresses might collapse the entire graph into large superclusters that are not actually controlled by a single user. We should be more safe in deciding about this. This could lose, for sure, in utility.

During the project I also tried to stress anonymity of the bitcoins flow with real users looking around in forums and so on but I have to admit that I get lost.

It is not so easy to cluster addresses belonging to the same user with a poor set of information such as the one owned.

Moreover the user is given as a single entity, but we don't have a guarantee about this. So we are not accurate also about this fact.

An user could change its addresses in the meantime, as with a sort of deadline to make us get errors in such analysis.

In any case, what we had is just a partial view of an already passed story.

Java Report

------------------------------------------------------------------------

Building neo4jID 1.0-SNAPSHOT

------------------------------------------------------------------------

--- exec-maven-plugin:1.5.0:exec (default-cli) @ neo4jID ---

lug 01, 2021 1:15:45 PM org.neo4j.driver.internal.logging.JULogger info

INFO: Direct driver instance 2003891312 created for server address localhost:7687

Pulizia DB vecchio

Salvataggio delle transazioni

Salvataggio degli input

Salvataggio degli output

Individuazione relazioni tra input e transazione

Individuazione relazioni tra output e transazione

Individuazione relazioni tra input e output

Rimozione transazioni e elementi collegati che non rispettano il corretto bilancio

Individuazione transazioni accettate con almeno 6 regole di conferma

Rimozione transazioni e elementi collegati non accettate con almeno 6 regole di conferma

Individuazione blocchi di appartenenza delle transazioni

Salvataggio dei blocchi con le corrispondenti relazioni

There are 0same transaction in different blocks.

Individuazione massimo UTXO per ciascun indirizzo

The highest associated UTXO value of: 3000000000000 belongs to transactionID 136084 of the blockID 88148 with outputID 165004 and address 134506

Creazione nodo dell'AWARD CLUSTER

Creazione relazioni col nodo dell'AWARD CLUSTER

Creazione relazioni per l'identificazione degli USERS CLUSTER

Lowest address: 134505 Tot UTXOs owned: 3000000000000 of which the max UTXO owned: 3000000000000

TransactionID sending the greatest amount of bitcoins: 139898

Length of the longest payment path: 378735

Found a false negative!

NOTE: Was found a false negative between these public addresses: 131073 and 131084

They refer to different transactions: 132909 and 132919

lug 01, 2021 1:19:37 PM org.neo4j.driver.internal.logging.JULogger info

INFO: Closing driver instance 2003891312

lug 01, 2021 1:19:37 PM org.neo4j.driver.internal.logging.JULogger info

INFO: Closing connection pool towards localhost:7687

------------------------------------------------------------------------

BUILD SUCCESS

------------------------------------------------------------------------

Total time: 03:55 min

Finished at: 2021-07-01T13:19:39+02:00

Final Memory: 14M/60M

------------------------------------------------------------------------