Cryptocurrencies (WS2021/22) Prof. Matteo Maffei

Project 1 – Analysing the Blockchain

Due *28.11.2021, 23:59* TU Wien

In this project you will perform some simple analysis on blockchain data. Overall, this project has a maximum of **100 points**.

Before you start with the project, carefully read this document!

Setting up Your System for the Project

In this project, we require you to produce SQL code that performs an analysis on blockchain data. To this end we want you to use PostgreSQL. You can download the version suitable for you operating system from the PostgreSQL website. In order to be able to test your code by performing database queries, you also need to run a local server. For Mac users we recommend Postgres.app.

Once you are done installing, create a new database (the name does not matter) and connect to it. Then execute the file `init_blockchain.sql` that you can download from TUWEL. This will initialize the database in a way you need it for the first exercise.

General Submission Preparation

For each of the following exercises your task is twofold:

Part 1: Understand the structure of the data that should be queried and develop a strategy for obtaining the data. Collect the ideas and strategies that you have developed for the mental part in a `.tex` file (please use the template provided in TUWEL). For each exercise, we expect the file to contain the following information (written in your own words):

- Which concrete criteria should the data satisfy?
- Which intermediate steps are performed in order to obtain the desired data?

In the `.tex` file, do not forget to insert your references if you have consulted material outside of the lecture and cite your sources if you have discussed with other students. Before you submit, compile the file, such that you send us the `.pdf` version only and not the `.tex` version. The naming conventions are detailed in the submission instructions at the end of this document.

Part 2: Write SQL queries modifying the initial database in the way requested in the exercises.

Remark Please keep in mind that the submission deadlines are strict. If you miss a deadline your submission will be graded with 0 points!

Groups

You are entitled to submit the project in groups up to *3 students*. Please put name and matriculation numbers of all group members in the report. Additionally, report on the tasks that each of the group members was involved in. In case of imbalances, the work distribution might affect the individual grades of the group members.

Grading

You will be graded based on your answers in the `.pdf` file and the quality of the submitted `.sql` files. Full points will be given in case the database queries produce the expected result and the explanation demonstrates that you have understood what kind of data should be queried and your queries coincide with your explanation. In case you do not give any explanation, you will receive 0 points. If you give unsatisfactory explanations, we subtract partial points.

Introduction

By now you should be fluent with the structure of Bitcoin transactions. This assignment will draw on your knowledge of the blockchain structure and de-anonymization techniques. You will be given a truncated data set of Bitcoin transactions starting from the genesis block and ending at height 100,001. The block reward was 5,000,000,000 satoshis (50 BTC) during this period.

While this is almost entirely real Bitcoin data and the code you develop for the assignment could be adapted to work on the entire blockchain, the data is slightly simplified and we have removed or modified some transactions. For this reason, you will need to work with this dataset to get the correct answers – using an externally parsed version of the blockchain will lead you to incorrect results. (Please do **not** download the real Bitcoin blockchain. This won't help you :).)

Data schema

For this project, we provide you with simplified blockchain data in the SQL format. The data given represents the main branch of the blockchain. More precisely, you will get three different tables of the following forms:

- `transactions`:

tx_id	block_id

  - `tx_id` is an id uniquely identifying the transaction
  - `block_id` is the id of the block that the transaction is part of

- `inputs`:

input_id	tx_id	sig_id	output_id

  - `input_id` is an id uniquely identifying the input
  - `tx_id` references the transaction that uses this input
  - `sig_id` holds the public key used for the signature in the case that a standard signature check should be performed. If a more complex script was used, it holds -1 and if the referred transaction is a coinbase transaction, it holds 0. (Note that this a simplification: instead of modeling the signature, we are simply storing the public key of that was used to decrypt it)
  - `output_id` references the output that is used as input

- `outputs`:

output_id	tx_id	pk_id	value

  - `output_id` is an id uniquely identifying the output
  - `tx_id` references the transaction that produces this output
  - `pk_id` holds the public key of the recipient of the output in the case that a standard signature check is performed. If a non-standard script is used, it holds -1. (Note that this is a simplification: instead of modeling the script that would check the signature, we are simply giving the public key that the signature would be checked against)
  - `value` holds the output value in satoshis

All columns contain `integer` values. For this reason we will in the following not further comment on types, but expect the columns of all tables we mention to be of type `integer`.

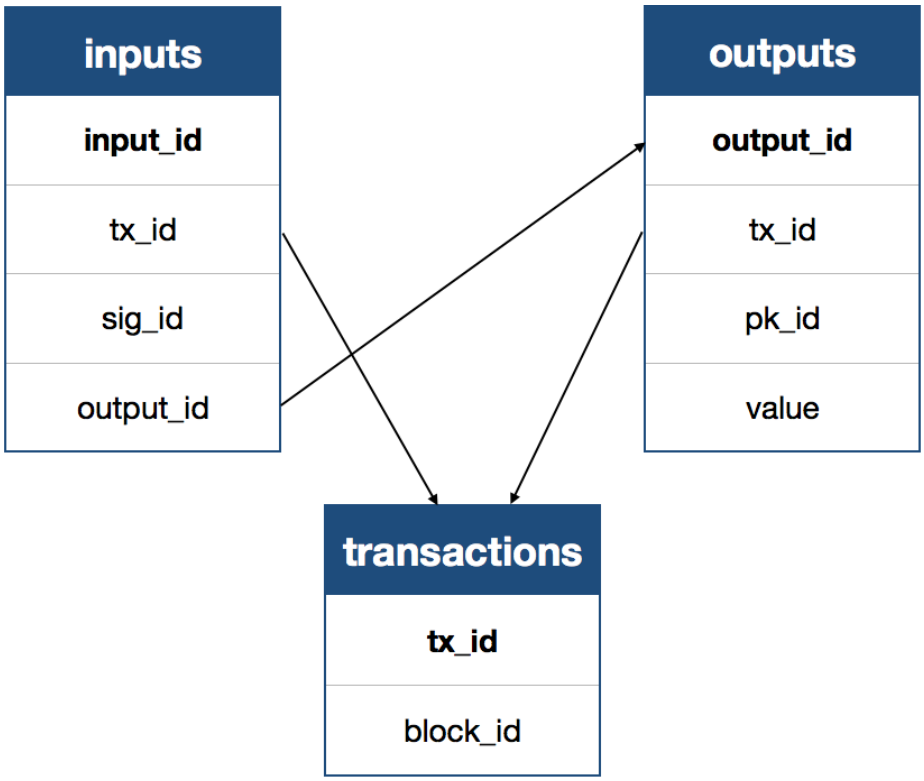

Figure 1: Visualization of the database scheme

Notice that in this model of the blockchain data several simplifications have been made:

- Instead of modeling scripts, we only distinguish the cases where a standard signature check is performed and those cases where it is not. In the case that a standard signature check is performed, the `sig_id` column of the `inputs` table holds the public key that was used for the signature and the `pk_id` column of the `outputs` table holds the public key of the output recipient. In the case of non-standard scripts being used, both columns hold the value -1. In these cases, from our data set, we cannot say whether the script execution was successful or not.

- Inputs reference outputs directly instead of referencing the transaction and specifying the offset of the output that should be spent

- We omit most of the bookkeeping information for transactions and blocks. In particular, we do not model the hash pointers to previous blocks and to the Merkle tree of transactions and consequently, we cannot easily obtain an order on transactions and blocks. For simplicity, we assume that the ids of blocks and transactions reflect their order. So, a block with a (numerically) lower `block_id` is assumed to precede a block with (numerically) higher `block_id` in the blockchain and a transaction with a (numerically) lower `tx_id` is expected to be listed before a transaction with a (numerically) higher id in the same block.

The data set already satisfies some consistency constraints that you do not need to check for. You can make the following assumptions on the database:

- the ids of `transactions`, `inputs` and `outputs` are unique
- all `inputs` and `outputs` refer existing `transactions`
- in the `inputs` table `output_id` has value -1 if and only if `sig_id` has value 0
- the provided tables do not contain any empty cells (`NULL` values)

In the TUWEL, you can download the following files:

- `init_blockchain.sql`: this file initializes a database for exercise A
- `init_blockchain_short.sql`: this file initializes a database for exercises B and C

In both cases, executing the file when connected to a fresh database, will result in populating tables satisfying the properties discussed before so that you can directly start performing SQL queries on the data. If you are not familiar with SQL, you can find a short summary of the most important features needed for this project at the end of this document starting from page 9.

Exercise A: Finding invalid blocks (50 points)

The database resulting from the initialization with `init_blockchain.sql` contains fifteen blocks that are invalid (note that these are not errors in the original chain, but artificially introduced ones). As we assume that our data set represents the main branch of the blockchain, it should hold that all blocks contained in the data set have been successfully validated by the clients using the block validation algorithm when added to the main branch (a write-up of the algorithm can be found here in the section `"block" messages`). For fifteen blocks however, this is not the case.

Find those fifteen blocks by writing SQL queries that insert the `block_id`s of the invalid blocks into the predefined table `invalid_blocks` that has a single column `block_id`. After running the `.sql` file you provide on the freshly initialized database, the `invalid_blocks` table should contain all invalid blocks.

Remarks

- We only consider those blocks invalid that would fail the checks in the block validation algorithm. Blocks that have transactions using outputs of invalid blocks are not invalid.
- Consider only the parts of the block validation algorithm that refer to components modeled by our data schema. A lot of the checks mentioned in the algorithm are clearly out of scope, as e.g., checking syntactic correctness or the proof of work. You can find a simplified version of the algorithm taking this into account on page 12.

Hint

If you found all invalid blocks, then the SHA256 Hash of their concatenation (without any spaces or other delimiters) of the block ids in ascending order should be

$$1ecd3e40266962decb372683cd9c0ae70e3ec7a3b6b18656a0d5d222f0350275$$

Exercise B: UTXOs (15 points)

In the following, you will work with a simplified and smaller version of the previous data set. To this end, connect to a fresh database and run the file `init_blockchain_short.sql`. In this new data set, you shall analyse the unspent transaction outputs (UTXOs). Recall that a transaction output is unspent if it is not used as input to a later transaction.

Write SQL queries insert values into the (predefined) result tables as follows:

1. The table `utxos` with columns `output_id` and `value` should contain all UTXOs as of the last block of the data set
2. The table `number_of_utxos` with column `utxo_count` should contain as single entry the total number of UTXOs
3. The table `id_of_max_utxo` with the column `max_utxo` should contain as single entry the id of the UTXO with the highest associated value

Exercise C: De-anonymization (35 points)

On the simplified data set, we want you to perform a de-anonymization attempt using the following two common heuristics:

- Joint control: addresses used as inputs to a common transaction are controlled by the same entity
- Serial control: the output address of a transaction with only a single input and output is usually controlled by the same entity owning the input addresses

For this exercise we assume that there is no mixing or obfuscation in place.

First, we want you to cluster the Bitcoin addresses according to these heuristics. As implementing a clustering algorithm in SQL is a bit cumbersome, we provide you with a function `clusterAddresses()` that performs the final clustering for you. This function however, operates on a table `addressRelations` with the two columns `addr1` and `addr2` that needs to be correctly populated. Create a an `.sql` file that performs the following steps. Make sure that it inserts your results to the specified tables when run on the freshly initialized database.

1. Insert all pairs of two addresses that belong to the same entity according to the idioms of joint control or serial control into the table `addressRelations`. You do not need to insert all reflexive or symmetric pairs as those will be generated automatically. Still, adding redundant pairs does not harm.

2. Use the `clusterAddresses()` function to perform the clustering. The function returns a table with columns `id` and `addr` where `id` contains an (artifical) id of a cluster and `addr` is an address belonging to this cluster. The table can be interpreted as a mapping from addresses to clusters (or entities controlling this address). Use the output of the function in order to insert values into the (predefined) result tables as follows (to this end, it might be helpful to save the result of the function in a new table):

   (a) The table `max_value_by_entity` with column `value` should contain as single entry the maximum total value of (unspent) satoshis controlled by one cluster (one entity).

   (b) The table `min_addr_of_max_entity` with column `addr` should contain as single entry the (numerically) lowest address of the cluster (the entity) controlling the most total (unspent) bitcoins

   (c) The table `max_tx_to_max_entity` with column `tx_id` should contain as single entry the transaction sending the greatest number of bitcoins to the cluster (the entity) controlling the most total (unspent) bitcoins

Remark

You can assume the table `utxos` from exercise B to be already filled correctly.

Submission Instructions

Please submit your solution by ***November 28th, before 23:59*** through TUWEL.
Upload a single `.zip` archive containing the following files:

- `report1.pdf`: The pdf containing the descriptions of your strategies and answers to textual questions.
- `a.sql`: The SQL file for exercise A
- `b.sql`: The SQL file for exercise B
- `c.sql`: The SQL file for exercise C

SQL

For analyzing the blockchain data, we want you to use SQL. In the following we provide a short overview on useful SQL constructs that might be beneficial when solving the exercises.

Queries

A basic SQL query has the following form:

```
SELECT ⟨column list⟩
FROM ⟨table⟩
WHERE ⟨condition⟩
```

So, the query

```
SELECT input_id, tx_id
FROM inputs
WHERE input_id > 10000
```

will return the information for the columns `input_id` and `tx_id` of those rows from the `inputs` table where the value in the column `tx_id` has a value bigger than 10000. The `WHERE` clause can also contain more complicated conditions involving logical connectives as `NOT`, `AND` and `OR`.

Joins

When the information of two tables should be combined, often a `JOIN` operation can help. The `JOIN` operation allows to combine the rows of two different tables that agree on a specified column.

For example the query

```
SELECT input_id, tx_id, block_id
FROM inputs JOIN transactions USING (tx_id)
```

returns a table that contains all input ids together with the ids of the transactions that they belong to and in addition also the ids of the blocks that those transactions belong to. In this simple case, the join should not blow up or shrink the size of the original `inputs` table as every input belongs to exactly one transaction and block. Looking at it from another perspective, the resulting table contains much more entries than the original `transactions` table, as transactions might have several inputs. But still all transactions from the original table should occur in the resulting table, as every transaction has at least one input.

This is not always the case. The query

```
SELECT output_id, input_id
FROM outputs JOIN inputs USING (output_id)
```

returns only those outputs that are referenced (spent) by some input (so for them there is a row in the `inputs` table with a matching `output_id`).

If still all outputs should be contained in the final query a so called `OUTER JOIN` can be used:

```
SELECT output_id, input_id
FROM outputs LEFT OUTER JOIN inputs USING (output_id)
```

In this case, the resulting table contains all entries from the `outputs` table even if there is no matching entry in the `inputs` table. In this case, the `input_id` column just carries the value `NULL`.

But also without joins, relations between tables can be exploited. For example by using powerful `WHERE` clauses:

```
SELECT output_id, tx_id
FROM outputs
WHERE EXISTS (
                SELECT *
                FROM inputs
                WHERE inputs.output_id = outputs.output_id)
```

This query selects the `output_id` and the `tx_id` for all those outputs that are referenced by at least one input in the `inputs` table. This is checked by requiring that the table containing only those inputs whose `output_id` column is the same as the one of the member of the `outputs` table that is currently considered is non empty. Note that in order to disambiguate the column names, it was here necessary to also specify the corresponding table that the column is taken from. The `*` in the `SELECT` clause of the query just means that all columns of the table shall be selected.

Grouping

Another useful feature for analysing data is grouping. The `GROUP BY` clause allows to collect the rows agreeing on the value of one (or several) specified column and to apply so called aggregate functions to them.

For example, the query

```
SELECT tx_id, count(*) AS input_count
FROM inputs
GROUP BY tx_id
```

counts the entries for each transaction in the `inputs` table. The `AS` construct can be used to rename a column and can also be useful in other contexts where conflicting column names should be avoided. Other interesting aggregate functions (besides `count`) are `sum` for summing up the values of the column specified in the argument per group and `max` and `min` for computing the minimum and the maximum of the group respectively. Be careful: In the case of grouping, the `SELECT` clause can only contain those columns used in the `GROUP BY` clause or aggregate functions.

When the results of aggregate functions should be used for filtering the outcome, a `HAVING` clause can be used:

```
SELECT tx_id, count(*) AS input_count
FROM inputs
GROUP BY tx_id
HAVING count(*) < 2
```

outputs a table containing the transaction ids and the number of inputs per transactions for all transactions that have less than two inputs. Be careful: the `WHERE` clause applies to the table before grouping while the `HAVING` clause applies to the result of grouping and can also contain aggregate functions.

Simplifying queries

For simplifying complicated queries, it is often a good idea to split big queries into subqueries using `WITH`.

For example, in order to obtain a table containing transactions together with the number of outputs and inputs they have, one could formulate the following query:

```
WITH transactions_with_inputcount AS
  (
    SELECT tx_id, count(*) AS input_count
    FROM inputs
    GROUP BY tx_id)
SELECT tx_id, input_count, count(output_id) AS output_count
FROM transactions_with_inputcount
     JOIN
     outputs
     USING (tx_id)
GROUP BY tx_id, input_count
```

Note that the intermediate table introduced in the `WITH` clause can be used in the later query. It is possible to define several intermediate tables within an `WITH` query by just sequencing them using a comma.

This example also shows how grouping can be used for several columns. In this case it is crucial not only to group using one column (e.g. `tx_id`) as this would not allow to select the other column (e.g. `input_count`) at all. But as there is anyway only one entry to the `input_count` column for each transaction id, grouping using both columns receives the desired result.

Inserting data

The only form of modification to the database that you need to perform, is to insert values into a table. The easiest form of doing so, is adding the results of a query (that matches the scheme of the target table) to a table.

```
INSERT INTO addressRelations
SELECT pk_id AS addr1, pk_id AS addr2
FROM outputs;
```

This query inserts the reflexive pairs for all addresses that got bitcoins transferred at some point into the `addressRelations` table:

This short introduction is by far not complete – eventhough we hope that it covers the main constructs needed for solving the exercises. For more details we refer you to the PostgreSQL documentation or to one of the numerous online tutorials (like this one or that one)

Block validation

In the following we present a shortened version of the algorithm presented in the Bitcoin wiki. To this end, we removed all points that do not apply to block chain model that we are using. Note that still we kept the original numbering so that you can compare to the orginal algorithm. Additionally, we added some notes (marked in green) and marked non relevant parts in gray.

6. First transaction must be coinbase (i.e. only 1 input, with hash=0, n=-1 with `sig_id` 0), the rest must not be

7. For each transaction, apply "tx" checks 2-4:

    2. Make sure neither in or out lists are empty

    4. Each output value, as well as the total, must be in legal money range (legal money range is from 0 to 21000000 BTC = 2100000000000000 Satoshis)

15. Add block into the tree (so append it to the block chain). There are three cases: 1. block further extends the main branch (we are always in this case, as the database only models the main branch); 2. block extends a side branch but does not add enough difficulty to make it become the new main branch; 3. block extends a side branch and makes it the new main branch.

16. For case 1, adding to main branch:

    1. For all but the coinbase transaction, apply the following:

        1. For each input, look in the main branch to find the referenced output transaction. Reject if the output transaction is missing for any input.

        4. Verify crypto signatures for each input; reject if any are bad (Verifying the crypto signature in our setting means to check whether the inputs `sig_id` corresponds to the public key (`pk_id`) of the output that it is referencing. In case of the use of non-standard scripts, both of these values should be -1)

        5. For each input, if the referenced output has already been spent by a transaction in the main branch, reject

        6. Using the referenced output transactions to get input values, check that each input value, as well as the sum, are in legal money range (legal money range as in 4.)

        7. Reject if the sum of input values $<$ sum of output values

    2. Reject if coinbase value $>$ sum of block creation fee and transaction fees