Cryptocurrencies (WS2021/22) Prof. Matteo Maffei

Project 1 – Analysing the Blockchain

TU Wien

Group 04

Our group consists of the following members:

Tobias Eidelpes, Mehmet Ege Demirsoy, Nejra Komic
01527193, 01641187, 11719704

Exercise A: Finding invalid blocks

For this exercise all invalid blocks contained in the database provided to us had to be found. While there is an official[1] algorithm which allows network participants to verify whether a block is invalid or not, the stripped-down version of the blockchain we received does not require all the steps. This stripped-down version of the algorithm thus specifies which constraints the data must satisfy:

1. All blocks which do not have the coinbase transaction as their first transaction are invalid. This will be achieved by creating a view which lists all coinbase transactions. Then we query the database for all first transactions of each block and check if that transaction is in the view of all coinbase transactions. If it is not, we reject the block and add it to the invalid list.

2. All blocks which contain transactions which do not have inputs or outputs are invalid. We split this task into two queries, one for checking if a block contains transactions with zero inputs and another one for checking if a block contains transactions with zero outputs.

3. All blocks which have transactions with an invalid output value or where the sum of all output values exceeds the legal money range are invalid. This task is split into two queries as well. One for checking if individual output values are outside of the legal money range and a second one for checking if the sum of all output values per transaction is outside of the legal money range.

4. Reject all blocks which have transactions with inputs that do not have a corresponding output. For this task we first create a view which finds all non coinbase transactions. The output of that query is then filtered for all inputs which are not part of a coinbase transaction (so the non coinbase inputs). Finally, the non coinbase inputs are joined with the outputs and rows containing `NULL` as their `value` indicate an invalid block.

5. All blocks which contain transactions where the input's `sig_id` field is not the same as the output's `pk_id` field are invalid. Since we are not interested in the coinbase transactions, the

[1] `https://en.bitcoin.it/wiki/Protocol_rules#.22block.22_messages`

query uses the non coinbase inputs again to join them with the outputs. If the two fields do not match, the block is invalid.

6. All blocks which have inputs for which there exist outputs which have already been spent are invalid. This task is split into three queries. First, we find all outputs which have more than one input. Second, for all the outputs found, we find the corresponding inputs where the output was first spent. Third, the two tables are combined such that blocks with outputs which have corresponding inputs that are not listed as the first spending occurrence, are marked as invalid.

7. All blocks containing inputs which are not in the legal money range are invalid. First, we construct a view which gathers all transactions and their corresponding sum of value for all inputs. All blocks containing input sums which are outside of the legal money range are marked as invalid. Second, we reuse the view of all non coinbase inputs and filter them for the ones which have an output value outside of the legal money range.

8. All blocks where the sum of input values is smaller than the sum of output values are invalid. This task allows us to reuse the view created earlier of all input sums. Additionally, the sum of output values is obtained similarly to the input sums. After joining both input sums and output sums, we can filter for blocks which have smaller input sums than output sums. Those blocks are invalid.

9. All blocks where the coinbase value is larger than the sum of the block creation fee and all transaction fees are invalid. This task is split into four queries. First, we create a view which shows all block ids and their coinbase values. Second, we need to know the sum of all input values per block. Third, we repeat that query for the sum of the output values per block. Lastly, these three tables are joined and all blocks which satisfy the constraint are invalid.

Finally, the invalid blocks are written to the `invalid_blocks` table and all duplicates are removed.

Exercise B: UTXOs

In this exercise we were given a smaller data set in comparison to the first exercise and we were expected to work on unspent transaction outputs which have the following constraint:

1. A transaction output is unspent if it is not used as an input to a later transaction.

The exercise further has the following constraints:

1. The table `utxos` with columns `output_id` and `value` should contain all UTXOs as of the last block of the data set. For this constraint we need to filter out the outputs from `outputs` table, whose `output_id`'s are not referenced in the `inputs` table. Thus we need a `WHERE NOT EXISTS` clause for the filtering.

2. The table `number_of_utxos` with column `utxo_count` should contain as single entry the total number of UTXOs. For implementing the solution of this constraint, we just need to count the number of `output_id` present in the `utxos` table from the previous constraint's implementation. `COUNT(output_id)` clause here is sufficient.

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. For getting the highest valued utxo, we need to order the `utxos` table in descending manner by the values. This would ensure that we have the highest valued utxo as the first entry. Thus by adding `LIMIT 1` clause, we get the top entry from the ordered results.

With each constraint, we insert the expected results into the given tables.

Exercise C: De-anonymization

In this exercise, a de-anonymization attempt was expected using the following two heuristics:

1. Joint control: addresses used as inputs to a common transaction are controlled by the same entity.

2. 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.

First part of this exercise was to insert all pairs of addresses into the table `addressRelations` satisfying the 2 constraints above. For this we first create 2 views each representing respectively the transactions that satisfy the above constraints. After that we use these tables to find pairs of addresses by performing (multiple) joins with `inputs` table and then insert the result into a temporary table called `tempRelations`. Since result contains reflexive and symmetrical pairs, we further up define additional queries to delete these pairs from the `tempRelations` table. With reflexive and symmetrical pairs deleted, we insert the `tempRelations` pairs into `addressRelations`, which concludes the first part.
For the second part, the function `clusterAddresses()` was provided for clustering the address pairs into entities with (artificial) ids. This function then returned a table with entity ids and the addresses belonging to these entities. First step is to save the results of the function into a temporary table. After this, following constraints have to be satisfied:

1. 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). To make our job easier, we save all the utxos with addresses, transaction ids, output_ids and values into a temporary table. We can then use this table in a join query with the table containing clusters by addresses. We then group entries by the entity ids and perform built-in `SUM` function on values and then call another built-in `MAX` function on the query result to obtain the maximum value.
   (Note: In our solution for readability purposes, we save the `SUM` results into a temporary table and query this table when we are querying for the max value.)

2. 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. To solve this, we first create a temporary table called `temp_max_entity` containing entity id, address and utxo values of all the addresses of the entity with maximum utxo value from constraint 1. Then we use this table to filter out all the addresses of this entity from cluster table, saving it into yet another temporary table called `max_entity_all_addresses`. As last step, we perform a `MIN` query on `max_entity_all_addresses` to satisfy the constraint.

3. 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. We start by creating a temporary table called `max_tx_value_to_max_entity`. The goal here is to save the value of the transaction, which sends the most amount of coins to an address of the max entry. For this query, we need to provide transaction id by joining `outputs` and `max_entities_all_addresses`. We construct this join query in a `WITH..AS` clause called `max_entity_join_outputs`. After that we use the same join query `max_entity_join_outputs`, but we additionally filter the result by the value in `max_tx_value_to_max_entity`, thus this leaves us with the desired transaction with the max value. The transaction id of this transaction is then inserted into the given table.

Work distribution

Tobias Eidelpes Code and report for Exercise A.

Ege Mehmet Demirsoy Code and report for Exercise C.

Nejra Komic Code and report for Exercise B.