Cryptocurrencies (WS2020/21) Prof. Matteo Maffei

Project 1 – Analysing the Blockchain

TU Wien

Group 15

Our group consists of the following members:

Greta Rausa, Luca Tasso, Tiago Fragoso
12007529, 12006420, 12005836

Exercise A: Finding invalid blocks

Our approach to this exercise was to follow the simplified block validation algorithm provided in the assignment and, for each point insert the blocks that violated the constraints. In fact, instead of inserting the blocks in the `invalid_blocks` table, we created a temporary table `invalid_blocks_temp` where we don't check if the block we are inserting was already deemed invalid. In the end, the distinct `block_id`s are inserted into the `invalid_blocks table`.

Upon quickly skimming the block validation algorithm, we found it beneficial to create 2 common views: `coinbase_transactions` and `non_coinbase_transactions`, as they would be reused throughout the algorithm.

A coinbase transaction must be the first transaction in each block and, in this case, its single input should contain `sig_id = 0`. So, we created a view which selects the transactions with only one input and `sig_id = 0`, by grouping the inputs by the transaction id and using the `HAVING` clause to inlude only transactions with one input and the sum of `sig_id` equal to 0. This sum is a nifty trick that allows us to do this in a single query because if there is only one input, then the sum of `sig_id` is the `sig_id` of the input itself. Then, we intersect the results of this query with the first transaction of each block, giving us a list of the coinbase transactions.

In order to get the inverse — non-coinbase transactions — we select all of the transactions and apply the `EXCEPT` operator to subtract the coinbase transactions calculated before.

The script snippet can be found below.

```
01 | CREATE OR REPLACE VIEW coinbase_transactions
02 | AS SELECT block_id, tx_id
03 | FROM (
04 |         SELECT tx_id
05 |         FROM inputs
06 |         GROUP BY tx_id
07 |         HAVING COUNT(input_id) = 1
08 |         AND SUM(sig_id) = 0
09 |      ) AS one_input_transactions
10 | JOIN transactions USING (tx_id)
11 | INTERSECT
12 | SELECT block_id, MIN(tx_id) AS tx_id
```

```
13 | FROM transactions
14 | GROUP BY block_id;
15 |
16 | CREATE OR REPLACE VIEW non_coinbase_transactions
17 | AS SELECT block_id, tx_id
18 | FROM transactions
19 | EXCEPT
20 | SELECT block_id, tx_id
21 | FROM coinbase_transactions;
```

Listing 1: Common coinbase transaction views

6. Invalid coinbase transactions

Starting by the first point in the algorithm, we must invalidate every block that either (1) does not have a coinbase first transaction and, (2) has a coinbase non-first transaction.

In order to tackle (1), we made use of the `coinbase_transcations` view created before and selected the blocks whose first transaction is not present in the view.

In order to tackle (2), we took advantage of the fact that we already found the blocks with a non-coinbase first transaction. So, we selected the blocks with more that one coinbase transaction (`sig_id = 0`). If the block has a coinbase first transaction, then this query will find the blocks where another coinbase transaction is present. If the block does not, it was already deemed invalid. Thus, both cases are covered.

```
01 | WITH first_txs AS (
02 |     -- Get the first transaction of each block
03 |     SELECT MIN(tx_id) AS tx_id, block_id
04 |     FROM transactions
05 |     GROUP BY block_id
06 | )
07 | INSERT INTO invalid_blocks_temp
08 | -- Get blocks that have a non-coinbase first transaction
09 | SELECT block_id
10 | FROM first_txs
11 | WHERE tx_id NOT IN (SELECT tx_id FROM coinbase_transactions);
12 |
13 | INSERT INTO invalid_blocks_temp
14 | -- Get blocks that have more than 1 coinbase transaction
15 | SELECT DISTINCT block_id
16 | FROM transactions JOIN inputs USING (tx_id)
17 | WHERE sig_id = 0
18 | GROUP BY block_id
19 | HAVING COUNT(tx_id) > 1;
```

Listing 2: Invalid coinbase transactions

7. "tx" checks

For the second point of the algorithm, there two conditions on which to invalidate blocks: (7.2) input or output lists are empty, and (7.4) each output or sum of outputs is outside the legal money range.

We started by finding the transactions with no inputs, joining the `transactions` and `inputs` tables, grouping by transaction id and using the `HAVING` clause to include only transactions with more than one input. We then select the `block_id` corresponding to each transaction and insert them

```
INSERT INTO invalid_blocks_temp
-- Get blocks that have transactions with no inputs
SELECT DISTINCT block_id
FROM transactions
JOIN inputs USING (tx_id)
GROUP BY tx_id
HAVING COUNT(input_id) = 0;
```

Listing 3: Blocks with one or more transactions with no input

Then, we selected the outputs whose value is outside the legal range (`value < 0 or value > 21000000 BTC`), using the value in *Satoshis*, and joined the result with the `transactions` table to get the `block_id` of each output.

```
INSERT INTO invalid_blocks_temp
-- Get blocks that have transactions with invalid output values
SELECT DISTINCT block_id
FROM transactions JOIN outputs USING (tx_id)
WHERE value < 0 OR value > 2100000000000000;
```

Listing 4: Blocks with one or more transactions with invalid output values

Finally, we combined the part of (7.2) with part of (7.4) to finish off — we selected the transactions whose (1) total output (`sum(value)` of all outputs) is outside the legal range (stated above) or (2) output list is empty. We did this by joining the `transactions` and `outputs` tables and grouping the rows by transaction id and used the `HAVING` clause to include the transactions matching (1) or (2).

```
INSERT INTO invalid_blocks_temp
-- Get blocks that have transactions with no outputs or invalid sum output
    values
SELECT DISTINCT block_id
FROM transactions
JOIN outputs USING (tx_id)
GROUP BY tx_id
HAVING COUNT(output_id) = 0 OR SUM(value) < 0 OR SUM(value) >
   2100000000000000;
```

Listing 5: Blocks with one or more transactions with invalid sum of output values or no outputs

16.1 Non-coinbase transactions

We noticed that for many 16.1.X steps, it would be useful to have a table with the inputs corresponding to each non-coinbase transaction, so we created a view named `non_coinbase_inputs` joining the `non_coinbase_transactions` view created in the beginning with the `inputs table`.

```
CREATE OR REPLACE VIEW not_coinbase_inputs AS (
    SELECT *
    FROM non_coinbase_transactions
    JOIN inputs USING (tx_id)
);
```

Listing 6: Non-coinbase inputs view

16.1.1 Inputs that reference invalid outputs

Regarding the next point the algorithm: a transaction is invalid (and subsequently its block) if one or more of its inputs references a missing output. We interpreted this as 2 possibilities: (1) the output referenced by the `output_id` in the transaction is not present in the `outputs` table, or (2) the output exists but it is created in a later (or same) transaction — i.e., `inputs.tx_id <= outputs.tx_id`.

We managed to solve (1) and (2) in a single query,by using a `LEFT OUTER JOIN` between the `non_coinbase_inputs` view and the `outputs` table, using the `output_id` column. This type of join will include every row from the table on the left and corresponding rows from the table in the right if possible. When there is no corresponding row in the table in the right, the columns are left as null. This allows us to check for rows with `NULL` values in the columns belonging to the outputs table, meaning that input references a missing output, solving (1). We used the `value` column but any of the columns in the `outputs` table would work (except for `output_id` which is common to both tables). In case there are no `NULL` columns, we also check if `non_coinbase_inputs.tx_id <= outputs.tx_id` to solve (2).

```
01 | INSERT INTO invalid_blocks_temp
02 | -- Get blocks containing transactions where inputs have no corresponding
       valid output
03 | SELECT DISTINCT block_id
04 | FROM non_coinbase_inputs
05 | LEFT OUTER JOIN outputs USING(output_id)
06 | WHERE value IS NULL OR non_coinbase_inputs.tx_id <= outputs.tx_id;
```

Listing 7: Blocks using inputs that reference missing/invalid outputs

16.1.4 Crypto signatures

In this case, we must reject blocks using inputs whose signature does not match the one in the related outputs. The solution is trivial and the blocks can be found by simply joining the inputs (`non_coinbase_inputs` view) with the `outputs` table and selecting the rows where `inputs.sig_id != outputs.pk_id`.

```
01 | INSERT INTO invalid_blocks_temp
02 | -- Get blocks where a input signature does not match the referenced output
       signature
03 | SELECT DISTINCT block_id
04 | FROM non_coinbase_inputs
05 | JOIN outputs USING (output_id)
06 | WHERE sig_id <> pk_id;
```

Listing 8: Blocks using inputs with mismatched input/output signatures

16.1.5 Double spend "attempts"

Step 16.1.5 is used to reject any blocks that contain double spend attempts — i.e., using as input an output already spent by another transaction in the main branch.

To find these blocks, we started by using a `WITH` clause to create a subquery that finds all of the `output_id`s that are used as input more than once, by joining our `non_coinbase_inputs` view with the `outputs` table and checking which outputs have more than 1 `input_id` associated. However, some of these are still valid — the ones that are spent first.

So, we used another `WITH` clause to find where these outputs were first spent, by selecting the minimum `input_id` for each of the `output_id`s found before.

Finally, we selected the `block_id` for the inputs that reference double spent outputs and are not the first reference of this output. By using `input_id` instead of `tx_id` in these queries, we make sure that if a single transaction references the same output more than once as an input, it is also rejected.

```
WITH double_spent_outputs AS (
-- Get outputs spent more than once
    SELECT output_id
    FROM non_coinbase_inputs
    JOIN outputs USING (output_id)
    GROUP BY output_id
    HAVING COUNT (input_id) > 1
), first_spent AS (
-- Get the input_id of the where the output was first spent
    SELECT MIN(input_id) AS input_id, output_id
    FROM double_spent_outputs
    JOIN non_coinbase_inputs
    USING (output_id)
    GROUP BY output_id
)
INSERT INTO invalid_blocks_temp
-- Get blocks containing invalid inputs due to double spend
SELECT block_id
    FROM double_spent_outputs
    JOIN non_coinbase_inputs
    USING (output_id)
    WHERE input_id NOT IN (SELECT input_id FROM first_spent);
```

Listing 9: Blocks with double spend attempts

16.1.6 Input values are in legal money range

In this step, we must reject blocks that use inputs whose value or sum of values is outside the legal money range (`value < 0 or value > 21000000 BTC`).

Firstly, we created a view named `input_sums` which contains the `block_id`, `tx_id` and the the sum of input values for that transaction because this will be reused going forward. This view simply aggregates all of the output values for each transaction using the `SUM` function.

After doing this, the query to find the blocks with invalid sums of input values is trivial — we can simply select the rows of the previously created view and include only the ones outside the legal money range.

To find the individual output values outside the legal money range, we joined the previously created view `non_coinbase_inputs` with the `outputs` table and included only the inputs outside the legal money range.

```
CREATE OR REPLACE VIEW input_sums
AS SELECT block_id, tx_id, sum
FROM transactions JOIN (
    SELECT non_coinbase_inputs.tx_id AS tx_id, SUM(value) AS sum
    FROM non_coinbase_inputs JOIN outputs USING (output_id)
    GROUP BY non_coinbase_inputs.tx_id
) AS inputs USING(tx_id);

INSERT INTO invalid_blocks_temp
-- Get blocks with transactions whose input sum is outside legal range
SELECT DISTINCT block_id
FROM input_sums
WHERE sum < 0 OR sum > 2100000000000000;

INSERT INTO invalid_blocks_temp
-- Get blocks with transactions whose input values are outside legal range
```

```
17 | SELECT DISTINCT block_id
18 | FROM non_coinbase_inputs
19 | JOIN outputs USING (output_id)
20 | WHERE value < 0 OR value > 2100000000000000;
```

Listing 10: Blocks with input values outside the legal money range

16.1.7 SUM(inputs) < SUM(outputs)

In this step, we must reject blocks containing transactions where the sum of inputs is less than the sum of outputs. As coinbase transactions are not not the subject of this check, any transaction where there outputs are greater than the input is invalid.

To find these, we made use of the previously created `input_sums` view and create an analogous view `output_sums` which aggregates the sum of output values of each transaction using the `SUM` function.

Finally, we joined the two views using the common `tx_id` column and included only the rows that violate the condition above.

```
01 | WITH output_sums AS (
02 |     SELECT block_id, tx_id, sum
03 |     FROM transactions
04 |     JOIN (SELECT tx_id, SUM(value) AS sum
05 |     FROM non_coinbase_transactions JOIN outputs USING (tx_id)
06 |     GROUP BY tx_id) AS outputs USING(tx_id)
07 | )
08 | INSERT INTO invalid_blocks_temp
09 | -- Get blocks with transactions whose sum(inputs) < sum(outputs)
10 | SELECT DISTINCT input_sums.block_id
11 | FROM input_sums
12 | JOIN output_sums USING (tx_id)
13 | WHERE input_sums.sum < output_sums.sum;
```

Listing 11: Blocks with total output greater than total input

16.2 Inflated coinbase value

A miner is allowed to create a coinbase transaction at the top of the block to collect the block reward plus the transaction fees. If the value of this transaction is greater than the sum of the block creation reward and the transaction fees, then this block is invalid.

To find these blocks, we used several `WITH` clauses to break down the problem.

Firstly, we found the coinbase value for each block, by joining the `coinbase_transactions` view with the `outputs` table to get the value of each transaction. Because a coinbase transaction can have multiple outputs, we aggregate the values using the `SUM` function, grouping by `block_id` — this is possible because the `coinbase_transactions` view already guarantees that there is only one transaction per block, so the grouping can be done by block.

Then, to calculate the transaction fees in each block, we calculate the sum of inputs and sum of outputs for each block.

We use a similar query for both inputs and outputs — joining `transactions` table with the `outputs` table, removing the coinbase transactions, aggregating the sum of values and grouping by block. However, for the input values, we use the outputs associated with each input, instead of the outputs associated with each transaction.

Finally, we join the three subqueries using `block_id`, giving us a "table" with columns: `block_id`,

`coinbase_value`, `input_values`, `output_values`. Then we select blocks that verify the formula:

$$coinbase\_value > block\_creation\_fee + transaction\_fees$$

where `block_creation_fee` is `50 BTC` and `transaction_fees` is `sum(inputs) - sum(outputs)`.

```
01 | WITH coinbase_values AS (
02 |     -- Get coinbase value for each block
03 |     SELECT block_id, SUM(value) AS coinbase_value
04 |     FROM coinbase_transactions
05 |     JOIN outputs USING (tx_id)
06 |     GROUP BY block_id
07 | ),
08 |  input_values AS (
09 |     -- Get sum(inputs) for each block
10 |     SELECT block_id, SUM(value) AS total_inputs
11 |     FROM transactions
12 |     JOIN inputs USING (tx_id)
13 |     JOIN outputs USING (output_id)
14 |     WHERE inputs.tx_id NOT IN (SELECT tx_id FROM coinbase_transactions)
15 |     GROUP BY block_id
16 |  ),
17 |  output_values AS (
18 |     -- Get sum(outputs) for each block
19 |     SELECT block_id, SUM(value) AS total_outputs
20 |     FROM transactions
21 |     JOIN outputs USING (tx_id)
22 |     WHERE tx_id NOT IN (SELECT tx_id FROM coinbase_transactions)
23 |     GROUP BY block_id
24 |  )
25 | INSERT INTO invalid_blocks_temp
26 | -- Get blocks where coinbase value > block creation fee + transaction fees
       :
27 | -- Block creation fee = 50 BTC; Transaction fees = sum(inputs)-sum(outputs
       ) of all transactions
28 | SELECT DISTINCT block_id
29 | FROM coinbase_values
30 | JOIN input_values USING (block_id)
31 | JOIN output_values USING (block_id)
32 | WHERE coinbase_values.coinbase_value > (5000000000 + (input_values.
       total_inputs - output_values.total_outputs));
```

Listing 12: Blocks with inflated coinbase value

Finally, we select the distinct `block_id`s from our temporary table `invalid_blocks_temp` and insert them into the `invalid_blocks` table.

```
01 | INSERT INTO invalid_blocks
02 | SELECT DISTINCT block_id
03 | FROM invalid_blocks_temp;
```

Listing 13: Invalid blocks insertion

Exercise B: UTXOs

The second exercise revolves around the unspent transaction outputs (UTXOs), the outputs that have not been used as inputs for another transaction yet. We assume that all the blocks in the chain are valid, since they belongs to main branch of the bitcoin blockchain, which guarantees that each output has been either correctly consumed by an input or that is still unspent. To find the UTXOs we select all of the outputs and exclude those that appear with their `output_id` in the table `inputs`. The `output_id` and the `value` of the resulting outputs are stored in the table `utxos`.

```
01 | INSERT INTO utxos
02 | -- Get unspent outputs
03 | SELECT output_id, value
04 | FROM outputs
05 | WHERE output_id NOT IN (
06 |     SELECT output_id
07 |     FROM inputs);
```

The total number of UTXOs can be obtained by counting the entries of the `utxos` table that we have just created.

```
01 | INSERT INTO number_of_utxos
02 | -- Get total number of unspent outputs
03 | SELECT count(*) AS utxo_count
04 | FROM utxos;
```

Similarly, we can find the id of the UTXO with the highest associated value using the `MAX` function on the `value` column of the table `utxos`.

```
01 | INSERT INTO id_of_max_utxo
02 | -- Get output_id of the UTXO with the highest associated value
03 | SELECT output_id AS max_utxo
04 | FROM utxos
05 | WHERE value = (
06 |     SELECT MAX(value)
07 |     FROM utxos);
```

Exercise C: De-anonymization

The third part, is about the de-anonymization of transactions. It will not give us a real identity behind each address, but we will be able to group them into clusters, and to piece together an history about each cluster. For helping us doing that, we will follow two heuristics rules: Joint control and Serial control.

Joint control

We know that addresses used as inputs to a common transaction are usually controlled by the same entity. In order to retrieve these entities we will check inside the Input table, looking for all of those addresses (`sig_id`) that share the same transaction id (`tx_id`), coupling them while avoiding associating an address with himself and finally putting the couples in the `addressRelations` table.

```
01 | INSERT INTO addressrelations
02 | SELECT I1.sig_id, I2.sig_id
03 | FROM inputs AS I1, inputs AS I2
04 | WHERE I1.tx_id = I2.tx_id
05 | AND I1.sig_id != I2.sig_id;
```

Serial control

The heuristic rule dictates that the output address of a transaction with only a single input and output is usually controlled by the same entity owning the input addresses. So first of all, we isolated the transaction ids which appears just once in the inputs and once in the outputs. We select then all the input and output addresses couples which appears in these transaction and insert them in `addressRelations`. We avoid the inputs addresses which are equal to 0 because they represent coinbase transactions.

```
01 | WITH in1_out1 AS (
02 | -- Get transactions with only 1 input and 1 output
03 |     SELECT tx_id
04 |     FROM transactions
05 |     JOIN inputs USING (tx_id)
06 |     JOIN outputs USING (tx_id)
07 |     GROUP BY tx_id
08 |     HAVING COUNT(inputs.input_id) = 1 AND COUNT(outputs.output_id) = 1
09 | ), in_addresses AS (
10 |     SELECT tx_id, sig_id AS in_addr
11 |     FROM inputs
12 |     WHERE inputs.tx_id IN (SELECT * FROM in1_out1)
13 |     AND sig_id <> 0
14 | ), out_addresses AS (
15 |     SELECT tx_id, pk_id AS out_addr
16 |     FROM outputs
17 |     WHERE tx_id IN (SELECT * FROM in1_out1)
18 | )
19 | INSERT INTO addressrelations
20 | SELECT in_addr, out_addr
21 | FROM in_addresses
22 | JOIN out_addresses USING (tx_id)
23 | WHERE in_addr <> out_addr;
```

So we are finally able to run the `clusterAddresses()` function to create the clusters. For ease of use we decided to store the result in a new table `clusters`.

```
CREATE TABLE IF NOT EXISTS clusters (id int, address int);
```

For the second part of the excercise we had to find values to put in the tables: `max_value_by_entity`, `min_addr_of_max_entity`, `max_tx_to_max_entity`. To achieve this, we created an ad hoc function, which first finds the id of the cluster that has the most unspent money, looking for correspondences of its addresses in a join between `outputs` and `utxos` tables. Afterwards we can easily fulfill subsequent requests, making some simple matches with the id we found.

```
DO $$
DECLARE max_cluster_id integer;
BEGIN
    -- Get id of cluster holding most unspent BTC
        SELECT c.id INTO max_cluster_id
        FROM outputs AS o, clusters AS c, utxos AS u
        WHERE o.pk_id = c.address
        AND o.output_id = u.output_id
        GROUP BY c.id
        ORDER BY SUM(o.value) DESC
        LIMIT 1;

        INSERT INTO max_value_by_entity
        -- Get maximum unspent BTC held by one single entity
        SELECT SUM(o.value) AS unspent_total
        FROM utxos AS u, outputs AS o
        WHERE u.output_id = o.output_id AND o.pk_id IN
                (SELECT address
                FROM clusters
            WHERE id = max_cluster_id);

        INSERT INTO min_addr_of_max_entity
        -- Get lowest address belonging to the entity holding the most
    unspent BTC
        SELECT MIN(c.address)
        FROM clusters AS c
        WHERE c.id = max_cluster_id;

        INSERT INTO max_tx_to_max_entity
        -- Get transaction sending the most BTC  to the entity holding the
     most unspent BTC
        SELECT tx_id
        FROM outputs AS o, clusters AS c
        WHERE c.id = max_cluster_id
        AND c.address = o.pk_id
        GROUP BY o.tx_id
    ORDER BY SUM(o.value) DESC
        LIMIT 1;

END $$;
```

Work distribution

We feel the work was **evenly distributed** and even though the tasks were split between the members, everyone reviewed and gave feedback on each other's work. Specifically:

Greta worked mainly on exercise C and reviewed exercises A and B.

Luca worked mainly on exercise B, helped debug and correct exercise C and reviewed exercise A.

Tiago worked mainly on exercise A and reviewed exercises B and C.

The report was written collaboratively, with each member writing the part where they worked mainly. Everyone proof-read the report.