

# UTXO Diffs

- ☐ work in progress defining UTXO Diffs
  - [Motivation](#)
  - [Solution](#)
  - [Terminology](#)
    - [TXO](#)
      - [UTXO](#)
      - [STXO](#)
    - [UTXO Set](#)
    - [Outpoint](#)
    - [UTXO Collection](#)
    - [Full UTXO Set](#)
    - [Diff UTXO Set](#)
    - [Diff Child](#)
    - [UTXO Diff](#)
  - [Diff Algebra](#)
  - [Restoring the UTXO Set of a Block](#)
    - [Motivation](#)
    - [Method](#)
  - [Adding a New Block](#)
    - [UTXO and Accepted Transactions Commitments](#)
    - [Updating Diff Children and Virtual](#)
  - [Example](#)

## Motivation

Kaspa uses the UTXO model. Full nodes maintains a current state of the UTXO Set. There are times when a node needs to know what the the UTXO Set looked like at a historical moment, e.g. when needing to validate a block from its point of view at the time it was created. Thus a node needs to be able to restore the UTXO Set to any moment until finality. Since the UTXO Set in Kaspa with its high transaction throughput can grow quite large, it is wasteful to save a full copy of the UTXO Set for each block.

## Solution

The solution involves having two types of UTXO Set objects:

- **Full UTXO Set** - a full copy of the UTXO Set of the [virtual block](#)
- **Diff UTXO Set** - a diff from the full UTXO Set, saved for each block

## Terminology

### TXO

TXO stands for Transaction Output (TX Output).

Each transaction consumes one or more inputs and produces one or more outputs. The inputs are essentially references to outputs of previous transactions that were unspent yet, called Unspent TX Outputs (UTXO). The inputs a transaction consumes must be valid (i.e., exist in the [past](#) of the block that includes the transaction, and unspent yet).

Transaction 1

Version	Inputs				Outputs				Locktime	Subnetwork ID
	Num Ins	In 1	...	In N	Num Outs	Out1	...	Out M		

Here, tx2.input\_N spends tx1.output\_1

Transaction 2

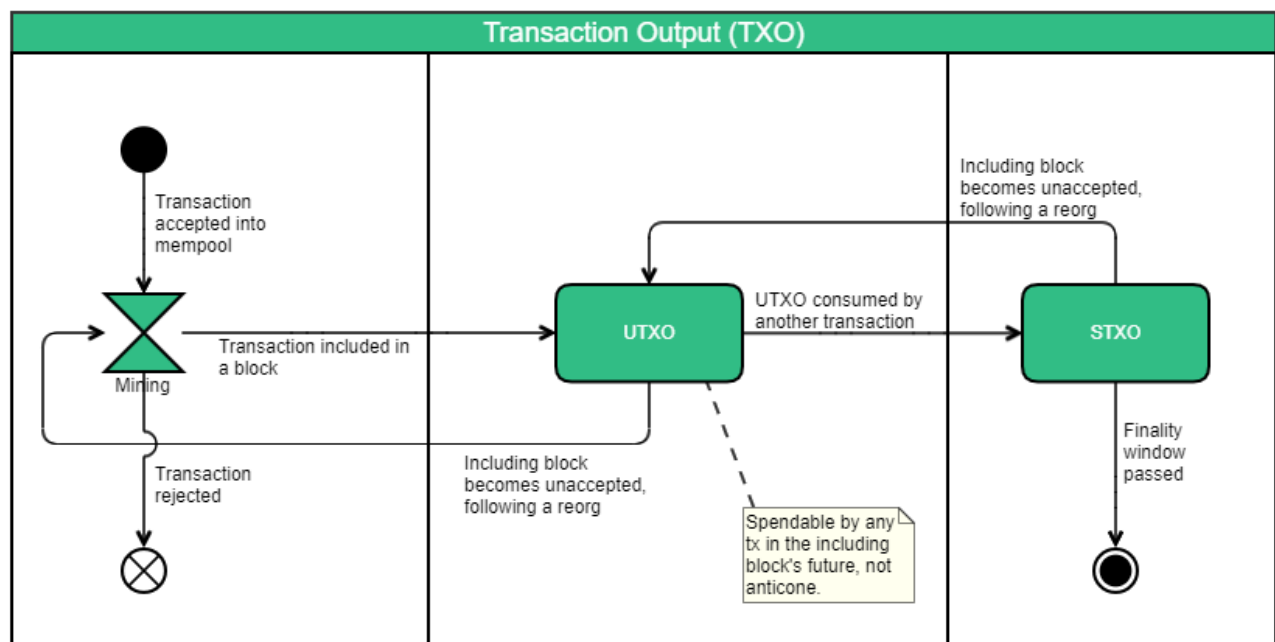
Version	Inputs				Outputs				Locktime	Subnetwork ID
	Num Ins	In 1	...	In N	Num Outs	Out1	...	Out M		

Time



UTXO cannot be partially consumed. It can only be entirely consumed. The sum of all the UTXO a transaction consumes must be greater or equal to the sum of the new outputs it produces. The remainder, if there is any, constitutes the fee, that the transaction creator offers miners for including it in a block.

To summarize, TXO is consumed and produced by transactions. It exists for [accepted transactions](#). Once it exists, it is first unspent (called UTXO), until it is spent (called STXO). It may become unspent again, during a reorg.



## UTXO

**UTXO** stands for Unspent Transaction Output (Unspent TX Output).

A UTXO is created once a [transaction is included in a block](#). It is then consumable (spendable) by any transaction in any block in the [future](#) of its [including block](#). Once a transaction spends a UTXO, that UTXO is considered an STXO and is removed from the UTXO Set.

## STXO

**STXO** stands for Spent Transaction Output (Spent TX Output).


A UTXO becomes an STXO once it is consumed by another transaction and that [transaction is accepted](#).

## UTXO Set

The UTXO Set is the set of all unspent transaction outputs.

It is all made of the set of all accepted transaction outputs (TXO Set) subtracted by the set of all accepted transaction outputs that were spent (STXO Set).

```
UTXO Set = TXO Set - STXO Set
```

 When referring to “the UTXO Set of a block B”, what is actually meant is the UTXO Set of Past(B), i.e. the UTXO Set known to B before B was created, and not including the transactions in B itself.

## Outpoint

An **outpoint** is a pointer to a transaction output. It consists of a sets of:

1. An immutable Transaction ID
2. An Index

## UTXO Collection


A **UTXO Collection** is a data structure used to represents a UTXO Set. It is a map from outpoints to transaction outputs.

```
UTXO Collection = map : pairs of (transaction id, index) --> TXO
```

## Full UTXO Set

The **Full UTXO Set** is a UTXO collection object containing the current state of the UTXO Set of the entire [blockDAG](#) known to the [virtual block V](#), i.e. `UTXO_Set(Past(V))`.

The Full UTXO Set exists only for the [virtual block](#).

 The Full UTXO Set is stored in memory, as well as in sequential order on the disk.

## Diff UTXO Set

A **Diff UTXO Set** is an object consisting of a `base` - a pointer to the Full UTXO Set, and a `diff` (UTXO Diff) - a difference from the Full UTXO Set.

```
Diff_UTXO_Set(B) = [base, diff]    -- or...  
Diff_UTXO_Set(B) = [*Full_UTXO_Set, UTXO_Diff]
```

A Diff UTXO Set object has an identical API to the Full UTXO Set object, allowing to perform the same operations on it as one would be able to perform on the Full UTXO Set.

## Diff Child

A **Diff Child** of a block B is defined as the chronologically-first [parent](#) of B.

Each block needs to maintain only one Diff Child.

When the Diff Child is the [virtual block](#), it is implemented as `null`.

**i** Note that the Diff Child link direction is opposite to that of blocks' parents, and that while a block may have several parents, each block has one and only Diff Child.

## UTXO Diff

A **UTXO Diff** of a block is defined as the needed information to obtain the UTXO Set of said block from the UTXO of its Diff Child.

UTXO Diffs are commutative, meaning they can be added and subtracted.

A **UTXO Diff** is implemented as two UTXO Collection objects:

1. `diffToAdd` - a [UTXO Collection](#) object containing the difference needed to add to the source [UTXO Set](#) in order to get to the destination UTXO Set.
2. `diffToRemove` - a [UTXO Collection](#) object containing the difference needed to remove from the source [UTXO Set](#) in order to get to the destination UTXO Set.

## Diff Algebra

We define the following two operations:

```
set.diffFrom(set) = diff
set.withDiff(diff) = set
```

Let A and B be UTXO Sets with the same `base`.

And let `c` be the difference in UTXO Set of A from B.

```
A.diffFrom(B) = c
A.withDiff(c) = B
```

**i** When using `A.diffFrom(B)`, the `base` part of A and B has to be the same, otherwise it will panic.  
When using `A.withDiff(c)`, the resulting UTXO Set, B, will have the same `base` as A.

The order matters. `A.diffFrom(B)` will return a diff `c`, that if applied to A will give B, and not the other way around. `B.diffFrom(A)` will return a different diff `c'`, that if applied to B will give A.

We define another operation:

```
Diff_UTXO_Set.meldToBase
```

Let `V'` be the Diff UTXO Set of a new virtual block from the current virtual block.

This method replaces the Full UTXO Set of the current virtual block with that of `V'`.

```
V'.meldToBase
```

## Restoring the UTXO Set of a Block

### Motivation

Being able to restore the UTXO Set of a block play a crucial part in validating a block. In order to validate a block's transactions, all of the inputs consumed by each of its transactions must have existed in the UTXO Set of said block at the time it was created.

**i** Future blocks could annul some or all of those transactions, but that does not invalidate the block. For the block to be valid, all its transactions must have been valid from the point of view of that block's [past](#).

### Method

Each block in the DAG has a route of Diff Children from itself to the virtual block. As mentioned, UTXO Diffs are commutative. In order to restore the UTXO Set of block B, the path of Diff Children from B to the [virtual block](#), V, is traversed, stacking references to the blocks along that route. Once reaching the virtual block, the diff childrens are popped from the stack and their UTXO Diffs are accumulated. A cumulative UTXO Diff is built this way containing the difference between the UTXO of B and that of the virtual block. The UTXO Set of Block B is then defined as a Diff UTXO Set, such that:

```
restoreUTXO(block B) =  
  B.diffFrom(V) = [base: *Full_UTXO_Set(V), diff: Agregation_of  
    (UTXO_Diffs from V to B)]
```

Likewise, the same algorithm can be applied generically to acquire the UTXO Diff between any block A and a block in its future B.

```
UTXO Diff of A from B =  
  A.diffFrom(B.diffFrom(V)) =  
    A.diffFrom(V) = [base: *Full_UTXO_Set(V), diff: Sum_of(UTXO_Diffs  
      from V to A)]  
  - B.diffFrom(V) = [base: *Full_UTXO_Set(V), diff: Sum_of(UTXO_Diffs  
      from V to B)]  
  
  =
```

## Adding a New Block

### UTXO and Accepted Transactions Commitments

When a new block B is added, its Diff UTXO Set is calculated in the following way:

1. Restore the UTXO Set of B's [selected parent](#).
2. Iterate over the blocks in B's [blues](#) in [PHANTOM order](#) (B's [selected parent](#) first then the rest of B's [blues](#) in [PHANTOM order](#)). For each block in B's blues:
  - a. Iterate over transaction included in the block. For each transaction T:
    - i. If and only if all of T's inputs are available in the restored UTXO Set:
      1. Subtract the outputs referenced by the inputs (by Transaction ID and Index) from the UTXO Set.  
*We do this, because the inputs do not contain the same fields as the outputs, and when calculating the UTXO Commitment of block B, we want to hash the fields in output.*
      2. Add T's outputs to the UTXO Set.
      3. Mark that transaction as [accepted by block B](#).
    - ii. Otherwise (i.e., if at least one of the inputs are not in the UTXO Set):
      1. Ignore that transaction and continue to the next one.
      2. Do not add it to the accepted transactions of block B

3. Do not remove any of its inputs from the UTXO Set and do not add any of its outputs to the UTXO Set.

## Updating Diff Children and Virtual

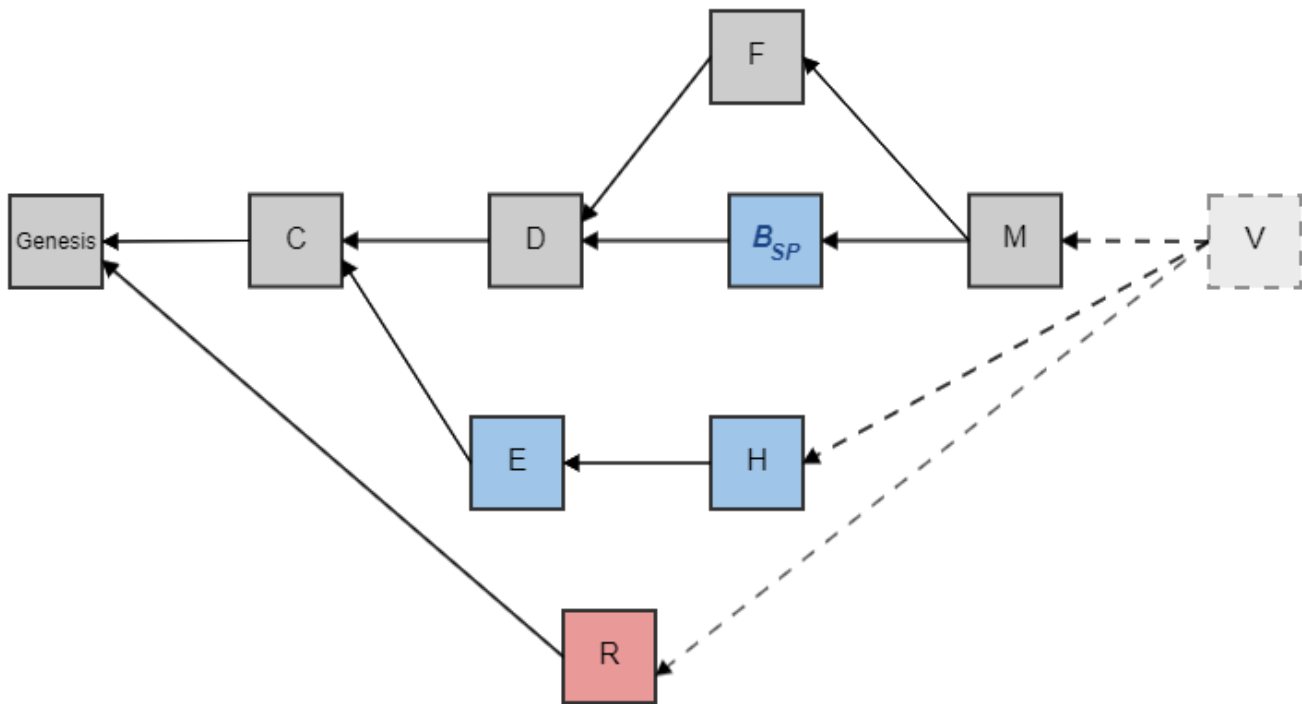
Once a new block B is added, the UTXO set of virtual V is updated in the following way:

1. Create a Diff Child and UTXO Diff for each of B's parents, that do not have a Diff Child yet.
2. Instantiate a new virtual block V', pointing to B and other [tips](#) (parents of V) as [parents](#).
3. Restore the UTXO Set of V' as a Diff UTXO Set from V (as described above in Restoring the UTXO Set of a Block).
4. Update the Diff Child and UTXO Diff of all new tips (parents of V'), using diff algebra, e.g. `tip_i.diffFrom(V')`.
5. Perform the switcheroo between V and V', using `V'.meldToBase` - it is a destructive operation that will overwrite the Full UTXO Set with the Diff UTXO Set of V'.

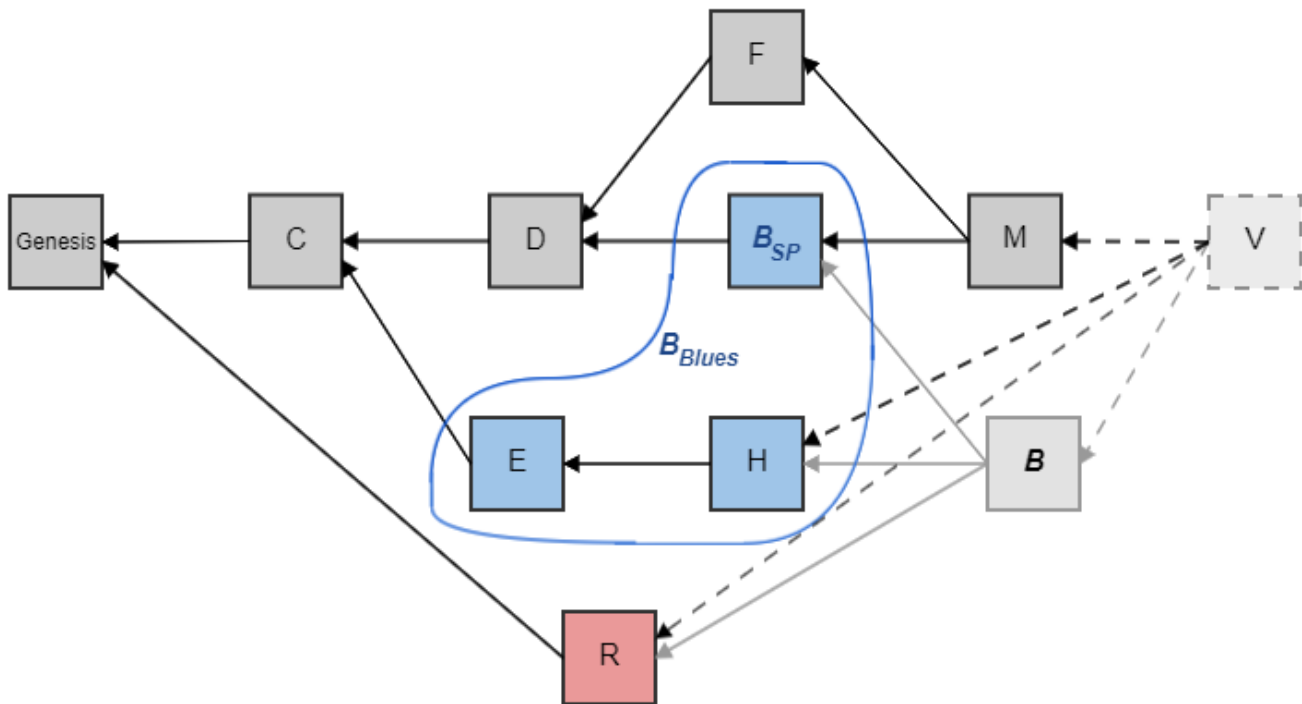
## Example

☐ A walk-through of how UTXO Diff data is maintained step-by-step, like in this be

Let us look at the following DAG.



The node wants to add block **B**. In order to do that, first, it needs to restore its UTXO Set, to figure out if its transactions are consistent with the UTXO Set at the time **B** was created.

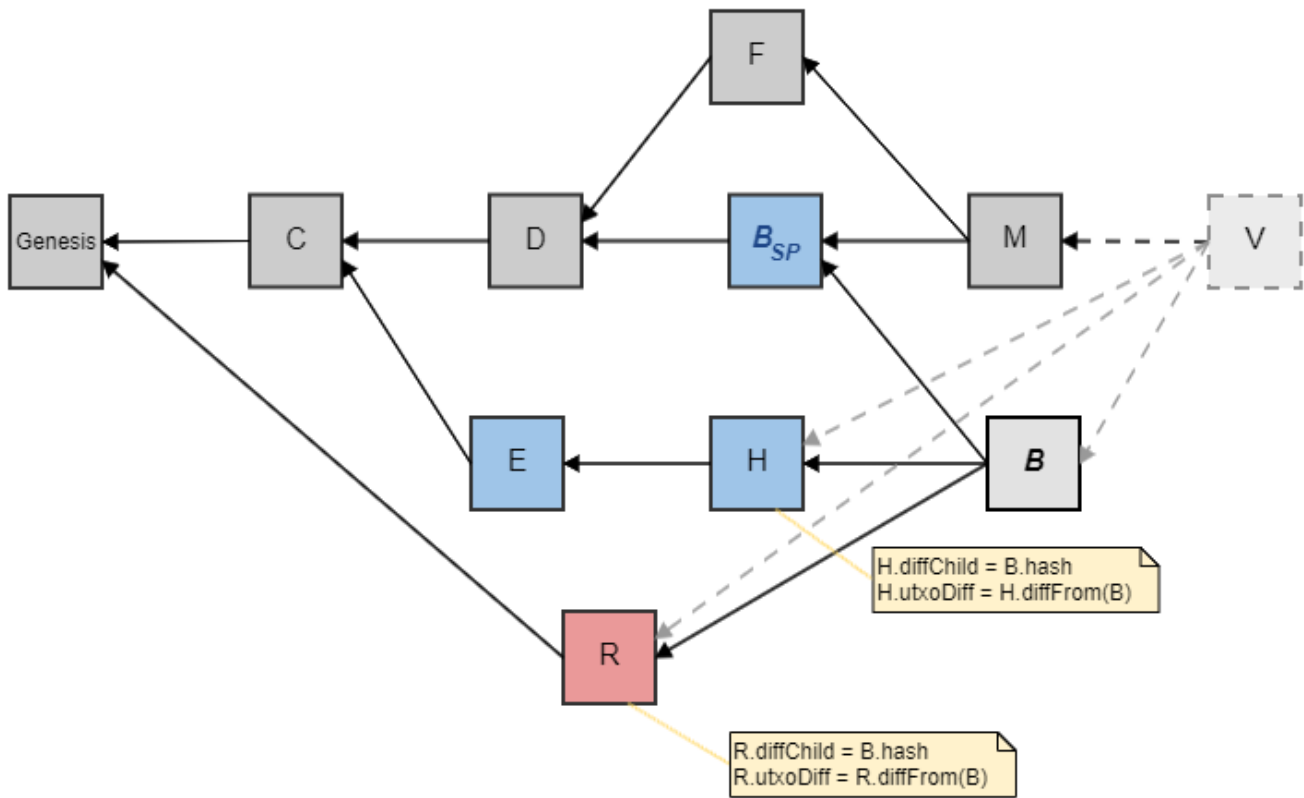


Jumping into this process:

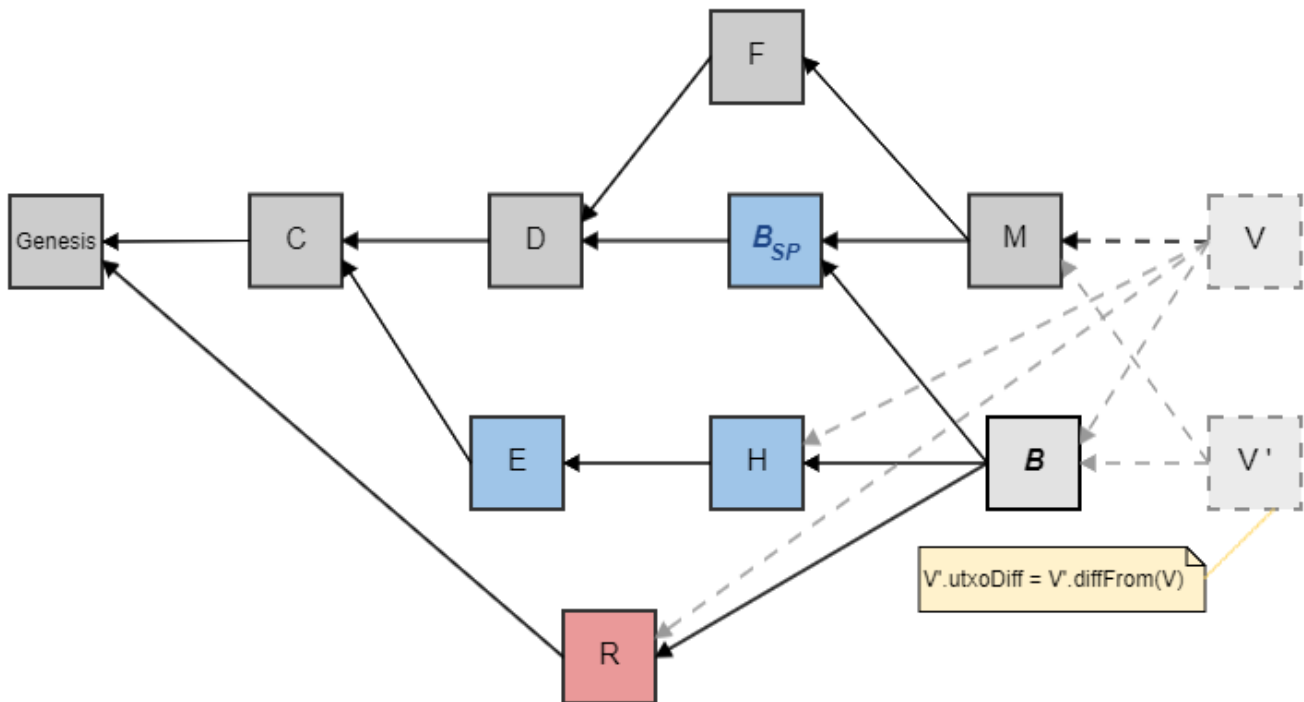
- First, the node restores the **UTXO Set** of B's **selected parent**, denoted **B.sp**.
- Then it attempts to add transactions from **B.Blues** according to **PHANTOM** order, i.e. first transactions from **B.sp**, then transactions from **E** and finally from **H** (note that we assume  $k=3$  and thus **R** is not in **B.Blues**). For each transaction attempted to be added, if all its **inputs** point to **outputs** that are available in the UTXO Set, then the transaction is **accepted**, the outputs it spends are removed from the UTXO Set and the outputs it produces are added to the UTXO Set.
- Lastly, it attempts to add **B**'s own transactions. They all need to consume outputs, that are available in the UTXO Set, that was reached at in the previous step.

**i** Note that **B** can include transactions that double spend inputs of transactions in **R**. The current implementation completely ignores red blocks.

Once the node has determined that block **B** can be added, the next step is to create a **Diff Child** and **UTXO Diff** for each of **B**'s **parents**, that do not have a Diff Child yet. Those are blocks **H** and **R**, whose Diff Child was *null* since they were previously only pointed at by the **virtual block**.



The next step is to create a new virtual block  $V'$  that points to  $B$  and other tips. The parents of  $V'$  are blocks  $B$  and  $M$ . Then, the UTXO Set of  $V'$  is restored.



Jumping into this process:

- First, the node restores the UTXO Set of  $V'$ 's selected parent – block  $B$ .

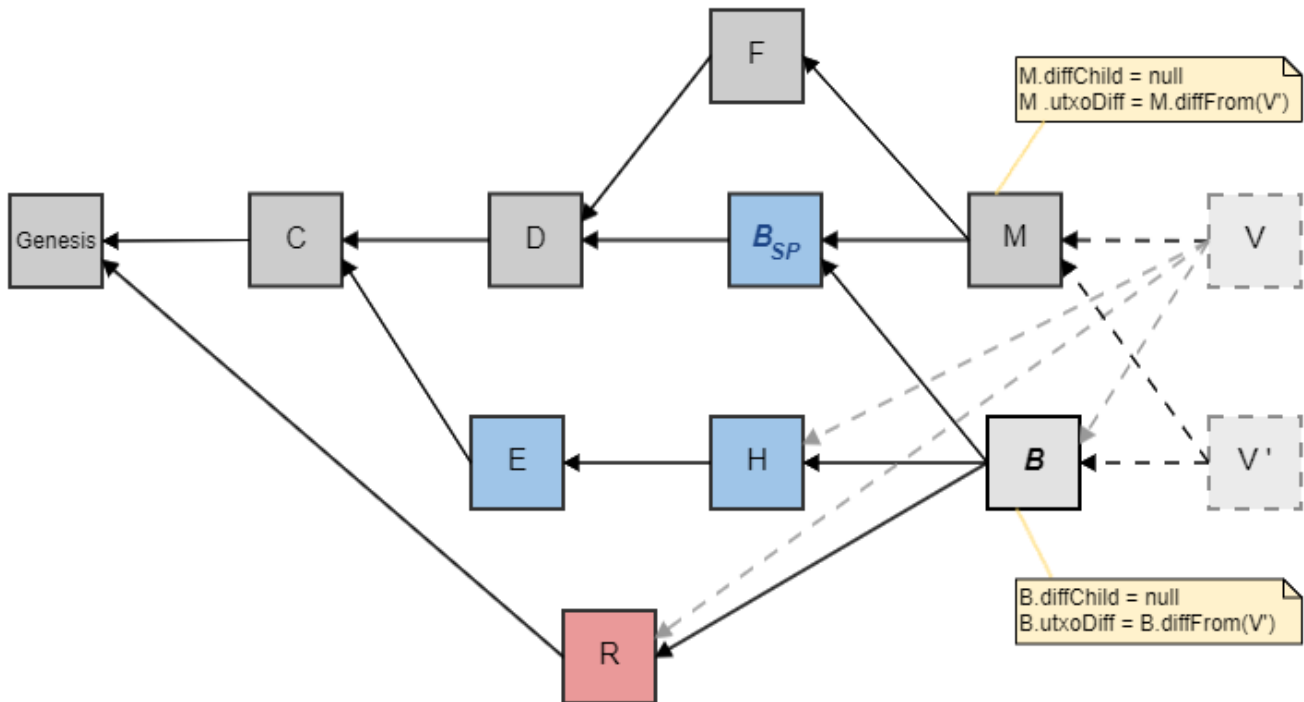


Block **B** has six blue blocks in its past, while **M** has five, hence the selected parent of **V'** is **B**.

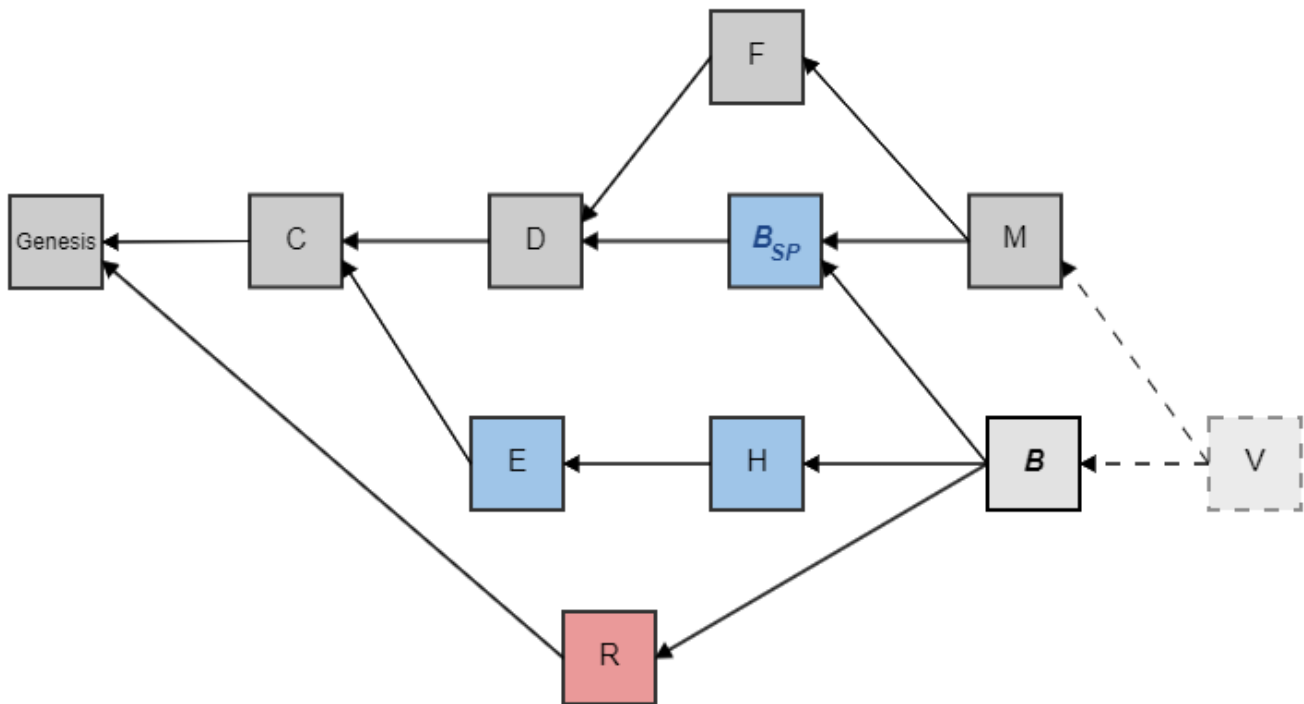
- Then the node attempts to add transactions from **V'.Blues** according to **PHANTOM** order, i.e. first transactions from **B**, then transactions from **M**. For each transaction attempted to be added, if all its **inputs** point to **outputs** that are available in the UTXO Set, then the transaction is **accepted**, the outputs it spends are removed from the UTXO Set and the outputs it produces are added to the UTXO Set.

Note that once **B** was added, **M** moved out of the selected parent chain, and **B** moved into the selected parent chain.

Once **V'** is added, the node updates the **Diff Child** and **UTXO Diff** of all new tips (parents of **V'**), using diff algebra.



The last step is to perform the switcheroo between **V** and **V'**, using **V'.meldToBase**.



Or more nicely, by arranging the blocks so that the selected parent chain is a straight chain in the middle, we get:

