Tezos Storage

Merkle Trees, git, context_hash

Big Picture

Our goal is to have a key value storage where the key is an array of strings, value is an arbitrarily long vector of bytes and the storage needs to implement git-like semantics, including history. The main constraint in implementation is computation of the hash of the whole storage, which needs to produce the same hash values as irmin does.

Operations

SET. Set key/value.

GET. Read value under key.

MEM. Check if value under a given key exists. Return bool.

COPY. Arguments *from* and *to*. For all keys with *from* as a prefix, create new keys/values with the *from* prefix replaced with *to*. Example: storage with a value [a, b] = 1. We perform COPY from [a] to [x]. Resulting storage has pairs [a, b] = 1 and [x, b] =1.

DELETE. Unset/remove a key or all keys given with with key as a prefix.

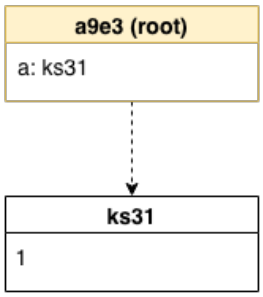
COMMIT. After we have done some writes on the storage with SET/COPY/DELETE, we can perform *commit* operation and get a hash of the current state. The commit operation is parametrised by time (u64), author's name (string) and a commit message (string). Blank strings are acceptable.

CHECKOUT. To get back in history of the storage state, we can *checkout* a state by hash values we get from the *commit* operation.

Implementation with Merkle trees

As it turns out, we can get all of these properties by using Merkle trees to model the data structure and RocksDB (or any other key-value store) to persist it. Let's illustrate with an example. We begin with an empty storage and perform a SET operation [a] = 1. We start off by computing a hash of the value 1: ks31. You can check this by computing Blake2b-256 hash of the string "1" and taking the first 2 bytes (4 hex digits). This will become the leaf node. Save it to RocksDB as a pair (ks31; 1). Now the node, which also happens to be the root of the node, that points to this value is a hash of a directory/map {a: ks31}, which equals to a9e3 (hash of "aks31").

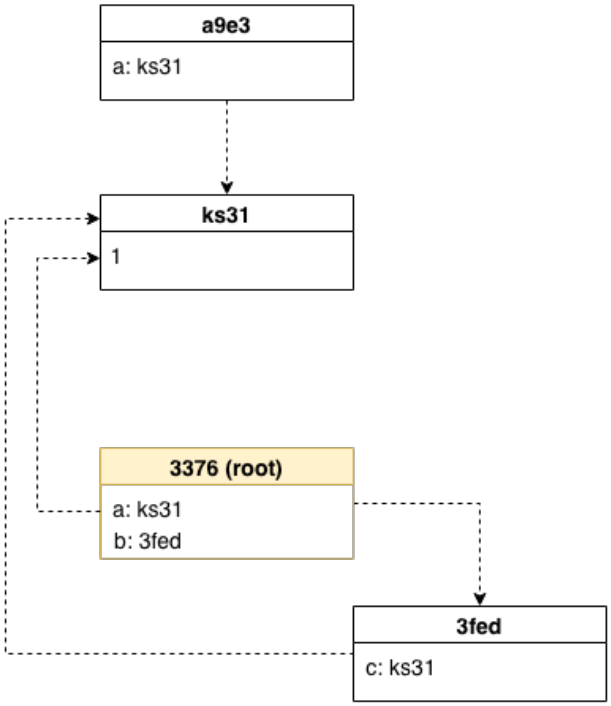
SET [a] = 1



DB State	
ks31	Value 1
a9e3	Tree {a: Value(ks31)}

Next, let's add a pair [b, c] = 1. Notice the value is the same as for the previous key. That means the hash of it is already computed and there is an entry with the hash and value in the database. Then we go backwards from the key to create the node that points to it: {c: ks31}. We hash it, its value is 3376 and store it in RockDB. We go 1 step back in the key to "b", which is the first position, therefore the root node. It will have two entries: {a: ks31; b: 3fed}. We hash that, too, and store it in RocksDB.

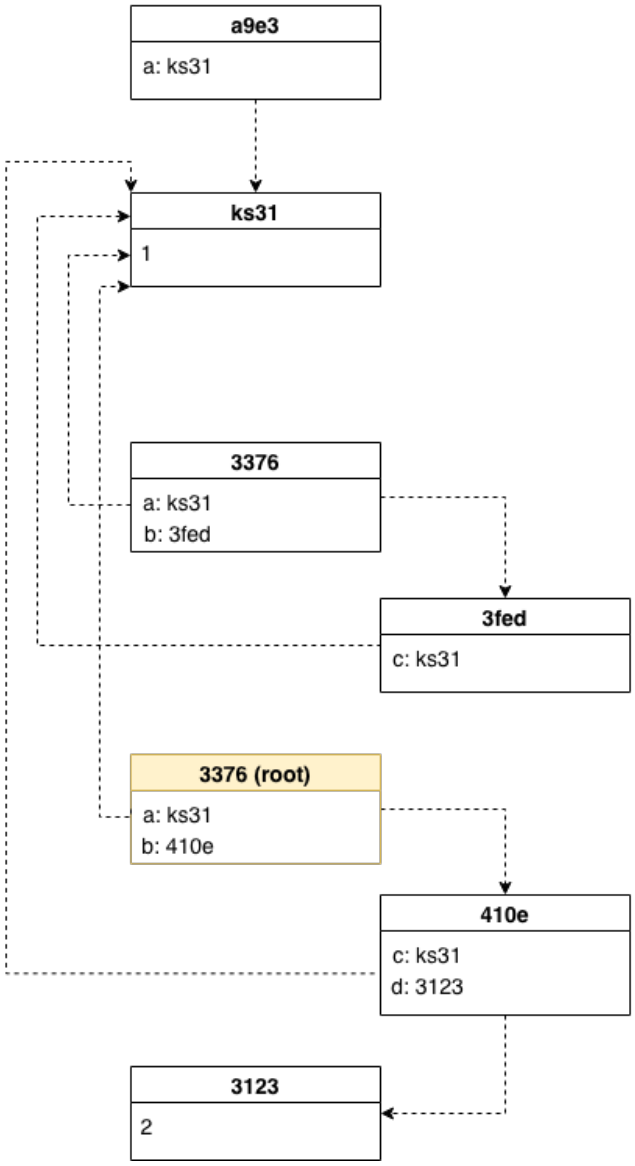
SET [b, c] = 1 (reuse existing value)



DB State	
ks31	Value 1
a9e3	Tree {a: ks31}
3376	Tree a: Leaf(ks31) b: Tree(3fed)
3fed	Tree {c: Leaf(ks31)}

As you can see, all the nodes from before are still there. If you wanted to know what the storage looked like before we performed the last operation, you could expand the node a9e3 and follow all the links to leaf nodes. Besides, notice the value 1 is stored only once in RocksDB. Merkle trees are efficient, in bigger trees whole nodes are left untouched and reused. Let's see what happens when we add [b, d] = 2. A new value and new subtree.

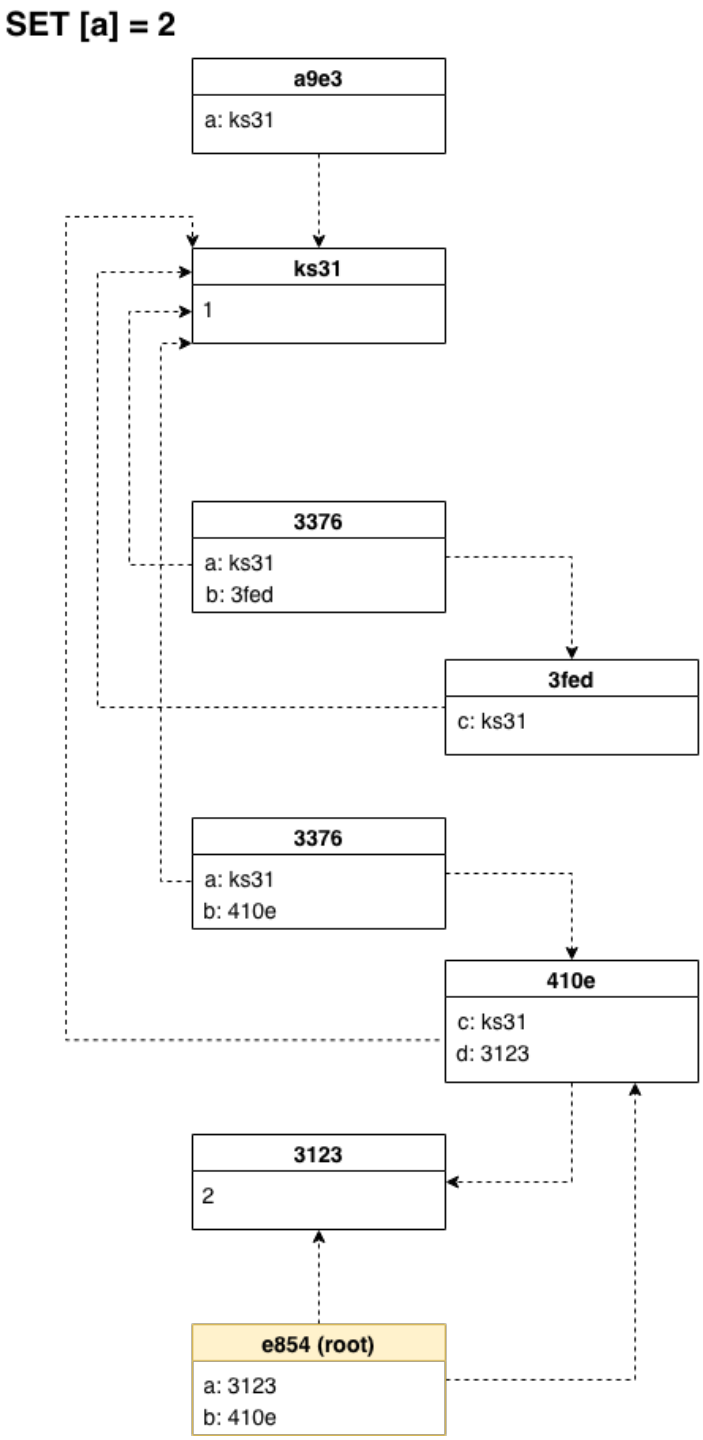
SET [b, d] = 2



DB State	
ks31	Value 1
a9e3	Tree {a: Value(ks31)}
3376	Tree a: Value(ks31) b: Tree(3fed)
3fed	Tree {c: Value(ks31)}
3376	Tree a: Value(ks31) b: Tree(4103)
4103	Tree c: Value(ks31) d: Value(3123)
3123	2

As always, se start by hashing the value (the number 2 this time) and storing it in the database under the its hash 3123. We work back from the key and create the subtree that will contain "d": {c: ks31; d: 3123}, its hash is 410e. Then the root must point to that node, too: {a: ks31, b: 410e}. As the dictionary/map in the node changes, the node's hash changes and we get a root node hash 3376.

In the previous examples, we always generated all the non-leaf nodes anew. The next step demonstrates the reuse of whole subtrees when a branch is left untouched. Let's update the value of [a] to 2. That means performing SET [a] = 2.



DB State	
ks31	Value 1
a9e3	Tree {a: Value(ks31)}
3376	Tree a: Value(ks31) b: Tree(3fed)
3fed	Tree {c: Value(ks31)}
3376	Tree a: Value(ks31) b: Tree(4103)
4103	Tree c: Value(ks31) d: Value(3123)
3123	2
e854	Tree a: 3123 b: 410e

The number 2 is already in the database, so there is nothing to do there. We didn't touch the subtree under [b] — no work to be done there, either. What changes is the root, that now needs to point *a* to a different hash: {a: 3123, b: 410e}. The node's contents changed, therefore its hash changed. The new hash e854 and it's the only new entry in RocksDB.

Computing hashes

Our examples compute hashes in the most simple way by concatenating the strings in keys and hashes. However, in Tezos the computation needs to conform to the implementation details of the *irmin* library. There are three kinds of objects hashes of which needs to be computed: values, nodes and commits.

BLOBS. These are the simplest. The input format is <length of data — 8 bytes><data>.

TREES.

- <length of items in node's dictionary — 8 bytes><items>
- Item: <INTERNAL_NODE_CODE or LEAF_CODE><length of string — 8 bytes><string bytes><length of hash — 8 bytes><hash bytes>.
- INTERNAL_NODE_CODE = FF-00-00-00-00-00-00-00 (8 bytes)
- LEAF_CODE = 00-00-00-00-00-00-00-00 (8 bytes)

Note that whether the hash points to leaf node or not is part of the hash input. See the "DB State" tables above: Values(hash) points to leaves, Tree(hash) to internal nodes.

COMMITS.

- <hash length — 8 bytes><tree hash bytes><number of parents — 8 bytes><parents><time — eight bytes><author's name length — 8 bytes><author's name bytes><message length — 8 bytes><message bytes>
- Parent: <length of hash — 8 bytes><hash bytes>