1.BackGround

The Ethereum community proposed an idea of the Path-Based Storage model, which is a big change to the storage. Currently, the feature is still under review and it is really complicated to understand and there is merely any document to describe it.

As a blockchain developer at NodeReal, a one-stop blockchain infrastructure and solution provider, I would take this opportunity to introduce this feature, so more people can understand it. Welcome to discuss more if there's any confusion.

2. Ideas

1. In the new path-based storage model, the trie nodes are saved in disk with the encoded path and specific key prefix as key, which means account address hash for account trie and storage address hash for storage trie. So the new updated path-based MPT will override the older one because of the same key for account trie and storage trie, it can not only achieve the goal of pruning but also can reduce data redundancy dramatically.
2. There's only a single version trie stored in the disk and will be updated in place whenever chain makes progress, and there will be only one version trie node belonging to the specific trie path be saved in the disk.
3. Use path-based trie node lookup, trie accessing and iteration is much faster than before, store a single version of state trie persisted to disk, and keep new tries(state/storage/account trie changes) in memory only.
4. The diff layer and disk layer structure can be represented by the following diagram, just like the snapshot tree, trie nodes belonging to different states are organized in different layers. There are at most 128 depth layers maintained, the bottom-most diff layer will be flushed into disk if there are too many nodes accumulated, also to serve reorgs
5. Support multiple version state access
6. Support in-memory node journaling for surviving restarts
7. Support state reverting by reverse diffs

+----------------+----------------+

| State Y | State Y' |

+----------------+----------------+

| State X | State X' |

+----------------------------------+

| ..... |

+----------------------------------+

| Bottom-most diff layer |

+----------------------------------+

| Disk layer(singleton trie) |

+----------------------------------+

1. Previously, trie.Database has a huge dirty node set which acts as the temporary container. Whenever tries are committed, dirty nodes are blindly inserted into this set and indexed by the node hash. However, in the new scheme, trie.Database organizes nodes in different layers, which means each layer must contain all dirty nodes corresponding to this state. Apparently, the committed nodes by a single trie is just a small part of them. So the commit procedure is changed a bit. Instead of pushing committed nodes blindly into dirty set, the committed nodes will be returned by trie, callers need to aggregate the results from multiple tries(account trie, storage tries) to a complete state diff for building a new diff layer

3. Crucial Parts of the Model

3.1. Current Geth Implement

- I’ll begin with the workflow of state transition and trie update in current Geth: Upon receiving one new block, the transaction inside the block body will be executed one by one and update the account data inside StateObject, will also update storage data if any smart contract is related. Then for every modified stateObject, firstly writing any storage changes in the state object to its storage trie, meanwhile, update the storage trie root. Secondly, commit the trie to the memory TrieDB. Finally, commit the memory TrieDB to levelDB for persist
- The image below showed the process for committing trie node into memory TrieDB, the initial MPT is image 4.1, for every node in the MPT, all the descendent children nodes will be collapsed down into hash node recursively as image 4.2(for simplify, only given two nodes whose descendent is hashnode as example.), then insert into the memory TrieDB
- Finally persist into disk db using hash as key and RLP format of collapsed node as value

3.2. Proposed Path-Based Mechanism

- The big difference is the way to persist into disk db, let’s compare the two different ways below:
- In the current levelDB, the way to persist is using the hash of trie node as key and the collapsed trie node as value as the left part in the below image
- In the path-based scheme, the key is the path of the trie node, which means the key will be the same for the same account and the same storage account. The value is the RLP version of the trie node
- In the current levelDB, the way to persist is using the hash of trie node as key and the collapsed trie node as value as the left part in the below image
- In the path-based scheme, the key is the path of the trie node, which means the key will be the same for the same account and the same storage account. The value is the RLP version of the trie node
- It’s very friendly and intuitive for inline state prune based on the new path-based model, let me clarify it using the below image and make some comparison as well, let’s assume the left MPT is the old state and the right is the new one, and the red are nodes with stale data, the green are the new version with the updated data
- In the current geth’s storage model, since the key in levelDB is the hash of tried node, it’s obvious that the hash between Node C and Node C’ is definitely different, although the Node C persisted into the levelDB, the Node C’ will still be persisted into as well, thus, all the red and green nodes will be into disk finally, and once persisted, there’s no any way for deleting them, the disk occupation will be larger and larger. The DB storage will be updated like below:
- In contrast to current geth storage logic, the newly path-based model stored using the path as key, so the key of Node C and Node C’ is the same, which is 0x1 in this example, 0x12f for Node D and Node D’ and so on like the below, the staled data of Node C, D, E will be overwritten by the same key with different value C’, D’ E’, so the staled data will be pruned naturally

[

vZqyNYCeirv-moiI0eb0tecqlAQTyRbz2mBtH9JVonn_Lqikt5TV0ouBVkxai8RJr9iXrV5MyfNR9C3I-effv6HGDiTn-nThVpLsyio73GVxvxLGqorXe6_2sIaff6MXcj46VhKqxozBmRYYxpK-Tos

1600×758 87.8 KB

](https://ethresear.ch/uploads/default/original/2X/8/862ccab3f73832555743cfa90020b464500129cf.png)

- WriteAccountTrieNode() function to write account trie node into database, the key is trieNodeAccountPrefix+hexPath of the trie node, the value is []byte format of the trie node
- WriteStorageTrieNode() function to write storage trie node into database, the key is trieNodeStoragePrefix + accountHash + hexPath of the trie node, the value is []byte format of the trie node
- The key and value of Deletion logic is same as the above Write

4. Some of Core Components Definition

- **Trie**

Add fields owner

: to identify the unique trie, for example, it’s contract address hash for storage trie

- **NodeSet**

NodeSet contains all dirty nodes collected during the commit operation. Each node is keyed by path.

- ## Trie Committer

- Commit all dirty nodes of a block in a single operation
- All the dirty nodes of a trie will be encapsulated in a struct called nodeSet

for return. Multiple nodeSets

can be merged together as a MergedNodeSet

. Eventually the MergedNodeSet

will be submitted to the in-memory database as a whole to represent the state transition from block to block.

1. The committer collapses all dirty nodes into hash nodes, no matter if it's a short node, full node. The key will be the node path, and returns with the modified nodeset
2. Store the hash node from step 3 above and add it into the modified nodeset => MergedNodeSet => db.TrieDB().Update(): perform state transition and add a new diff layer
3. Committer will collect all dirty nodes in the trie and replace them with the corresponding node hash, the nodes will be collapsed down into hash nodes. The difference with traditional Geth is that all collected nodes will be encapsulated into a nodeSet for return, and the the trie node path will be recorded within the nodeSet
4. ## SnapShot Interface (Logically, can be disk layer, diff layer…)

Snapshot is the extension of the Reader interface which is implemented by all layers(disklayer, difflayer). This interface supports some additional methods for internal usage.

- ## SnapDataBase

Similar to the current Geth's hashDatabase, snapDataBase is introduced by the path-based model. SnapDatabase is a multiple-layered structure for maintaining in-memory trie nodes. It consists of one persistent base layer backed by a key-value store, on top of which arbitrarily many in-memory diff layers are topped. The memory diffs can form a tree with branching, but the disk layer is singleton and common to all. If a reorg goes deeper than the disk layer, a batch of reverse diffs can be applied to rollback. The deepest reorg can be handled depending on the amount of trie histories tracked in the disk. At most one readable and writable snap database can be opened at the same time in the whole system which ensures that only one database writer can operate disk state. Unexpected open operations can cause the system to panic.

- ## Snap-Layer Tree

- This is a logic layer, which is a group of state layers identified by the state root, including diff layer, disk layer etc…
- A new snapshot can be added into the tree

- ## Snap-DiskLayer

- DiskLayer is a low level persistent snapshot built on top of a key-value store
- The given bottom-most diff layer will be merged into this diskLayer once commit

- ## Snap-Journal

- Load journal
- Can Load snap disk layer and reconstruct
- Load snap diff layer and reconstruct

- ## SnapTrieHistory

- Trie history records the state changes involved in executing a corresponding block. The state can be reverted to the previous status by applying the associated trie history. Trie history is the guarantee that the system can perform state

rollback, mostly for purposes of deep reorg. Each state transition will generate a corresponding trie history (Note that not every block has a state change, e.g. in the clique network or post-merge where no block reward rules exist). Each trie history will have a monotonically increasing number act as its unique identifier. The trie history will be written to disk (ancient freezer store) when the corresponding diff layer is merged into the disk layer. At the same time, the system can prune the oldest histories according to config. How does state rollback work? For example, if the system wants to roll back its state to the state n, it needs to ensure all histories from n+1 until the current disk layer are all existent, then apply the histories in order

Disk State ↑ +------------+ +---------+ +-----+ +---------+

| Init State |----> | State 1 |----> | ... |----->| State n |

+------------+ +---------+ +-----+ +---------+

+-----------+ +------+ +---------+

| History 1|----> | ... | ----> |History n|

+-----------+ +------+ +---------+

- At the time that the bottom-most diff layer merged into(commit), the snapTrieHistory will construct the trie history for this bottom-most diff layer, also will prune the stale histories from the disk with the given threshold

- # SnapDifflayer

- SnapDiffLayer represents a collection of modifications made to the in-memory tries after running a block on top. The goal of a diff layer is to act as a journal, tracking recent modifications made to the state that have not yet graduated into a semi-immutable state.
- A new snapDiffLayer can be created by snapDiskLayer on top of existing snapshot
- Keep 128 diff in the memory, persistent layer is 129th