Garbage collector for account state data

Aeternity Crypto Foundation
by
Karol Skočik
November 2019

Abstract In this paper we show how the garbage collection of account state nodes works in Aeternity Node and summarize the results.

Motivation

A chain in operation over several years accumulates a lot of historic data. Not all types of data the chain stores, bear the same level of significance to the functionality of the ecosystem. Sometimes the costs of keeping some types of data around overweights its utility to the users of the blockchain.

In particular, it is questionable if holding on to a history of all account balances at the expense of several gigabytes of space makes sense. While stored contract or channel may be useful many years after creation, such a statement doesn't hold for account balance history.

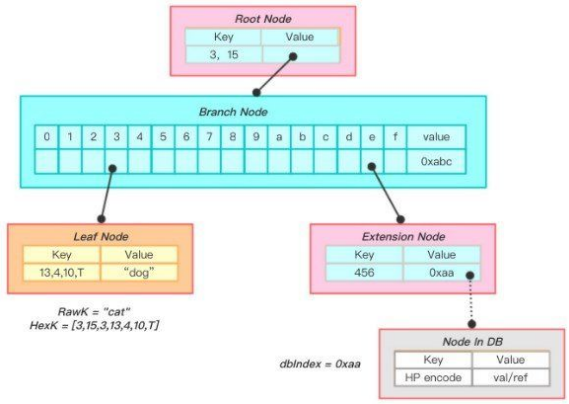
Persistent storage

Every piece of data governed by the consensus algorithm needs to be stored in a cryptographically authenticated data structure. In a cryptocurrency software, such authenticated data structure mostly takes the form of a *Merkle-Patricia Tries*.

Besides features specific to cryptography, the trie provides us with the abstraction of a key-value storage. Every object managed by a Aeternity node can be identified by a 32-bytes long byte array (hash) - a key to the respective Merkle-Patricia Trie. A value in the trie is a serialized key-value mapping of the elements describing the object.

Types of nodes in MP-Trie

The Patricia part of the trie organizes keys in a prefix manner. Sharing of the partial prefix(es) of the keys significantly lowers the space requirements of the trie and improves key path traversal time up to the leaf node keeping the serialized value of object identified by hash.



(1) Types of nodes in trie

The nodes are of four types:

- Null (terminator)
- Leaf (contains object value)
- Branch (16 way switch pointing to different continuations of key path)

- Extension (exactly one continuation of key path of arbitrary length)

As outlined above, when reasoning about space in Merkle-Patricia Trie, a single key-value pair doesn't contribute to the occupied space of the trie proportionally to the sum of its key and value sizes. In a populated trie, the traversal from the root to the leaf may need up to 16 lookups.

The non-leaf nodes, forming the path from the root to the leaf node, significantly contribute to the occupied space of the trie.
Fetching of the value from trie needs to traverse a sequence of branch and extension nodes of variable length.

Node types frequency

We used Aeternity node synced to the mainnet to conduct the following measurements.

The Merkle-Patricia Trie for accounts at height 165361 has 65379 nodes:

branches	16186	24.75%
extensions	564	0.86%
leaves	48629	74.38%

We can see that at this height, the node recognizes 48629 unique addresses.

If we look at the whole history of accounts, the numbers are more interesting:

branches	21394629	79.63%
extensions	52796	0.19%
leaves	5419221	20.17%

During the whole operation of Aeternity chain 26866646 nodes were written to the account state table. Over 165361 generations 5419221 were additions or changes to accounts.

Important insight here is that only 20.17% of all nodes in database table are actual values of serialized accounts. Almost 80% of all nodes are branches.

Since branch node is large when compared to leaf or extension (roughly - it's an array of 17 elements) - we can conclude that the vast majority of data bloat can be accounted to branch nodes.

It would be useful to have a direct correlation of object size and space occupied by this object when inserted to the trie. Unfortunately, it is not possible to come up with a reliable and exact measure of how much adding of one key value pair to MP-trie contributes to the overall space occupied by the trie. The branch and extension nodes are shared among leaf nodes, and placement of new "navigating" nodes in the trie ultimately depends on the values of the keys (hashes) already present in the trie.

Configuration

When reasoning about purging historic records via garbage collection, we need to state how long history we want to retain. For Aeternity node, we chose to keep history of last 500 generations by default (setting of *history* parameter can be overridden in configuration of the node).

Having history parameter set to 500 means that we will collect all reachable nodes of most recent 500 tries keeping the account state data. Accounts created earlier are still reachable from recent generations, but previous values of the same accounts with different balances will be purged.

Another configuration parameter is *interval* specifying the number of generations between two subsequent garbage collector runs.

The configuration is by default implicit and optional - for a GC to run, a node operator doesn't need to change the configuration file. In case tuning of the parameters is needed, user can make the configuration explicit by inserting following section to the configuration file:

```
chain:
      ... other chain config options
...
      garbage_collection:
            enabled: true
            interval: 50000
            history: 500
```

Design

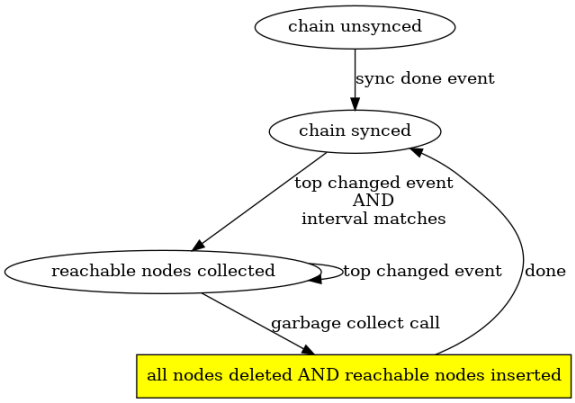
Every useful implementation of Merkle-Patricia Tries provides functionality which allows us to visit all nodes from root. We can utilize this feature and collect all reachable nodes - the ones we want to keep.

Merkle-Patricia Trie is immutable data structure - we don't change the trie by destructive modification of a node record in database. Instead, we create a new path from root to the added/changed value - a sequence of branch and extension nodes, sharing as many node records by referencing as possible.

This immutability allows us to inspect the tries and collect reachable nodes asynchronously.

GC finite state machine

The operation of garbage collector is best described as transition between states of a finite state machine.



(2) GC state transitions

In the *chain unsynced* state, the GC process waits until chain is synced with other peers. Unless node is restarted, it can not fall out from chain synced to chain unsynced state.

After receiving *sync done event*, the GC process waits again in *chain synced* state until it is time to run the collection, given the interval from configuration. With default interval set to 50000 generations, that may take roughly 3 months.

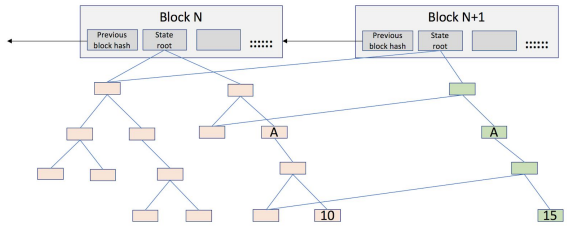
When the interval between GC runs matches, the process starts to collect reachable nodes, using earliest generation as a base.

EarliestGen = max(CurrentGen - History, 0)

The MP-trie of earliest generation is scanned in full - all reachable nodes are

visited and stored to transient cache. We will use this cache as a source of reachable nodes in the final phase of the garbage collection.

While the earliest generation is scanned in full, MP-tries of all subsequent generations can scan only differences to MP-trie from previous generation. The scanning starts from the root, but when already seen node is encountered (stored in transient cache), further scanning which would explore that node is stopped.



(3) Trie referencing previous generation

Image (3) illustrates the fact that MP-trie in generation Block N + 1 references only nodes of MP-trie from generation Block N, and doesn't reference to nodes of MP-tries from Block N - 1 or earlier. That is a useful property which allows us to mark a particular MP-trie as a *base*. If we collect all reachable nodes from this base MP-trie, all subsequent tries reference nodes from base MP-trie or newer.

After collecting all reachable nodes from earliest to current, the GC process is in *reachable nodes collected* state. In this state the GC process monitors changes of the top of the chain, and scans new reachable nodes of the latest generation. The GC process is ready to execute the final part of the garbage collection.

The *conductor* process in Aeternity node drives the progress by managing mining and writing records to the database. When the node is *not the leader* of the current generation and last added block is key-block, the conductor process calls the GC process to perform the swap of node records in the account state table. If the GC process is in *reachable nodes collected* state, the swap is executed.

The swap happens in *all nodes deleted AND reachable nodes inserted* state. Execution in this state is the only blocking part of the garbage collector functionality. It destructively modifies the account state table, where we delete all node records first and insert the reachable ones from the transient cache. After insertion of the nodes, the cache is purged.

Results

The first run of the garbage collector is the most important, as it removes unreachable nodes accumulated since the start of the chain. Testing on the mainnet, at generation 165424, revealed the following insight:

- Last 500 generations encompassed 85872 reachable nodes
- Collection of these nodes took 106.8 seconds on consumer grade SSD
- Deletion phase took 0.47 seconds
- Insertion phase took 2.75 seconds, writing 85906 nodes

The blocking part - swap phase - **took 3.22 seconds**.

A reader could notice that insertion wrote 85906 but scanned only 85872 nodes. The remaining 34 nodes were added to the transient cache after the *reachable nodes collected* phase finished,

as while the GC process was scanning recent generations, the chain was progressing.

The size of the account state table **dropped from 11GB to 24MB** - dropping 99.76% of data.

The frequency of node types (measured several generations later after garbage collection at generation 165485) looks as follows:

branches	35255	39.22%
extensions	598	0.66%
leaves	54029	60.11%

The Merkle-Patricia Trie for accounts at height 165485 has 65387 nodes, the frequency of node types is identical to MP-trie before garbage collection:

branches	16188	24.75%
extensions	564	0.86%
leaves	48635	74.38%

Conclusion

We have described how garbage collection of accounts works in Aeternity node. We believe that ordinary user should have the garbage collection enabled, unless she wants to retain the whole history of account changes. This may be useful for analysis of balance changes over time, where the operations *GetAccountByPubkeyAndHeight* and *GetAccountByPubkeyAndHash* are helpful. Their utility is proportional to the size of the history they can operate with.

References:

Implementation:
https://github.com/aeternity/aeternity/blob/GH-2845-accounts-state-gc/apps/aecore/src/aec_db_gc.erl
Merkle-Patricia Trie:
https://github.com/ethereum/wiki/wiki/Patricia-Tree