Nethermind client: 3 Experimental Approaches to State Database Change

Nethermind

Follow

Nethermind.eth

--

Listen

Share

In 2023, the Nethermind client team started developing two

State Database change approaches!

While fine-tuning these, our explorations led us to identify a third approach.

Users will benefit

from:

- higher validator rewards
- improved RPC performance & node maintenance
- simplified hardware requirements

We also hope to optimize and potentially remove the need for pruning and open doors for better archive nodes and snap sync servers.

Let’s dive into all 3!

Paprika

The first approach is the Paprika project —

an experimental storage for Nethermind, featuring a custom implementation of the Patricia tree. With Paprika, we want to address several things:

- Path-based access.

Here, the goal is to let nodes quickly access account data and storage slots, such as when retrieving or updating an account balance or a storage cell. With this approach, we managed to cut down on the time the EVM spends on data retrieval.

- Make Merkleization a pluggable component.

Merkleization is implemented as an independent component, separate from the rest. Although it needs to store and retrieve data from the underlying database, this is done in the final phase of processing, when the state root hash needs to be calculated.

- Reorganizations handling and block dependencies.

Paprika uses finality to execute the final data flush to the database. It stores changes from the non-finalized blocks in memory, predominantly in an unmanaged manner to reduce overheads like object management and garbage collection (GC). This approach looks promising and allows us to have a natural caching behavior for recently written data. This is particularly useful in the Merkleization of the state, where it helps by keeping the root parts in memory.

Paprika Github

GitHub - NethermindEth/Paprika: An experimental storage for Nethermind, removing the whole Trie…

An experimental storage for Nethermind, removing the whole Trie abstraction and acting as a

Trie and a database at once…

github.com

Path-Based Storage

The Path-Based Storage approach

aims to change the way that underlying data is organized at the DB level (RocksDB). Instead of using the hash of each Merkle-Patricia Trie node as a database identifier, the path to the node is used. The benefits of this approach are:

- shorter data access time as reading leaf nodes of MPT does not require traversing the trie — affecting block processing and JSON-RPC handling
- no pruning requirement because old data is naturally overwritten when changes happen

+ Plus, this opens the possibility for an efficient snap server implementation, making it fast and easy to fetch ranges of data from the database.

Path-based Storage Github

Flat / path based state layout by damian-orzechowski · Pull Request #6499 ·…

This is a draft PR with flat / path based implementation of state in NM to show the scope of the changes. This may be…

github.com

Halfpath

Interestingly, this approach emerged when optimization methods from the blocks, receipts, and headers databases were adapted for the state database. The team didn't anticipate that these methods would be effective when applied to the State DB!

The Halfpath approach

aims to boost the performance of the existing State Database without making major changes to the codebase — things like memory pruning, full pruning, and syncing code at a high-level stay mostly the same.

Here's what happens

- Under the hood, this method prefixes the key of the nodes with the trie path to that node, and this has a big impact.
- Even though it still doesn't let you access value without going through the trie, the way it arranges data helps with cache hits and speeds up block processing by almost 50%. Plus, it makes the database more compressible, shrinking its size by about 25%.
- Also, because we now have the path down to the storage layer, and different paths no longer share the same node, we can work on improving in-memory pruning by removing the existing node that is no longer in use in real time. This method is about 90% effective, slowing down database growth.
- In a 12-day test run, the uncompressed database size only grew by 1.28%, compared to the usual 14.62%.

That means we can go longer between full prunings.

Halfpath Github

Perf/HalfPath state db key by asdacap · Pull Request #6331 · NethermindEth/nethermind

Prefix state db entry with part of path Inspired by one of the guy I interviewed recently, what if we keep the whole…

github.com

Perf/improved inmemory pruning by asdacap · Pull Request #6439 · NethermindEth/nethermind

note this PR is more of a proof of concept to see if its viable We have seen #6331 reduces database growth by about 50%…

github.com

Conclusion

Builders & testers, we invite you to give these approaches a go and share your thoughts with us on Discord in #client-support!

While the first 2 approaches are still in development, the Halfpath approach will be implemented in the next Nethermind client release.

About Nethermind

Nethermind is a team of world-class builders and researchers. We empower enterprises and developers worldwide to access and build upon the decentralized web. Our work touches every part of the web3 ecosystem, from our Nethermind node to fundamental cryptography research and infrastructure for the Starknet ecosystem.

If you're interested in solving some of blockchain's most difficult problems, visit our

[job board

](https://www.nethermind.io/open-roles)!