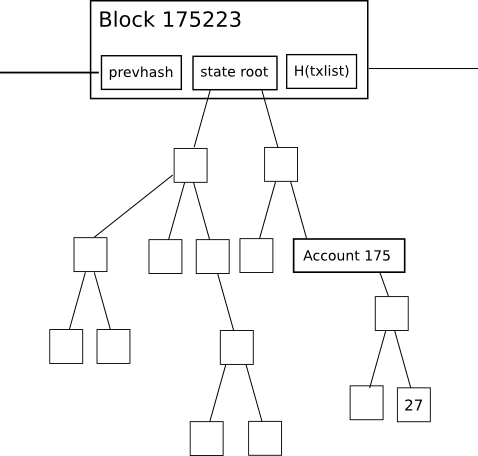
**[State Tree Pruning](https://blog.ethereum.org/2015/06/26/state-tree-pruning/)**

<https://blog.ethereum.org/2015/06/26/state-tree-pruning/>

**One of the important issues that has been brought up over the course of the Olympic stress-net release is the large amount of data that clients are required to store; over little more than three months of operation, and particularly during the last month, the amount of data in each Ethereum client’s blockchain folder has ballooned to an impressive 10-40 gigabytes, depending on which client you are using and whether or not compression is enabled. Although it is important to note that this is indeed a stress test scenario where users are incentivized to dump transactions on the blockchain paying only the free test-ether as a transaction fee, and transaction throughput levels are thus several times higher than Bitcoin, it is nevertheless a legitimate concern for users, who in many cases do not have hundreds of gigabytes to spare on storing other people’s transaction histories.**

First of all, let us begin by exploring why the current Ethereum client database is so large. Ethereum, unlike Bitcoin, has the property that every block contains something called the “state root”: the root hash of a [**specialized kind of Merkle tree**](https://github.com/ethereum/wiki/wiki/Patricia-Tree) which stores the entire state of the system: all account balances, contract storage, contract code and account nonces are inside.

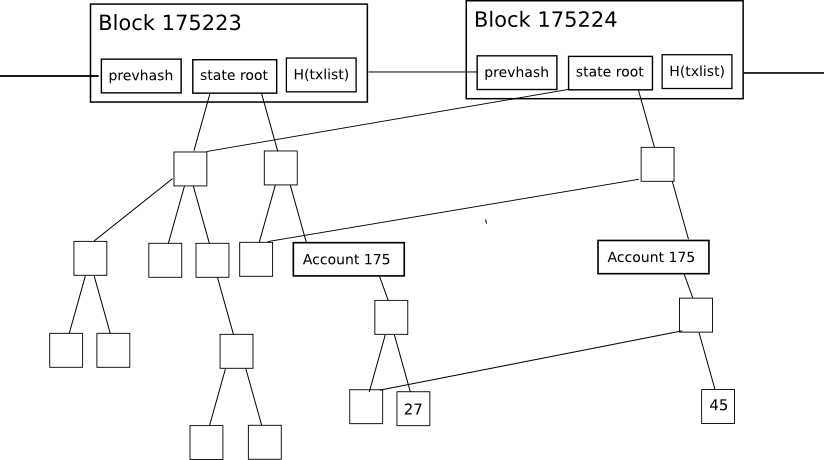


The purpose of this is simple: it allows a node given only the last block, together with some assurance that the last block actually is the most recent block, to “synchronize” with the blockchain extremely quickly without processing any historical transactions, by simply downloading the rest of the tree from nodes in the network (the proposed HashLookup [**wire protocol message**](https://github.com/ethereum/wiki/wiki/Ethereum-Wire-Protocol) will faciliate this), verifying that the tree is correct by checking that all of the hashes match up, and then proceeding from there. In a fully decentralized context, this will likely be done through an advanced version of Bitcoin’s headers-first-verification strategy, which will look roughly as follows:

1. Download as many block headers as the client can get its hands on.
2. Determine the header which is on the end of the longest chain. Starting from that header, go back 100 blocks for safety, and call the block at that position P100(H) (“the hundredth-generation grandparent of the head”)
3. Download the state tree from the state root of P100(H), using the HashLookup opcode (note that after the first one or two rounds, this can be parallelized among as many peers as desired). Verify that all parts of the tree match up.
4. Proceed normally from there.

For light clients, the state root is even more advantageous: they can immediately determine the exact balance and status of any account by simply asking the network for ***a particular branch*** of the tree, without needing to follow Bitcoin’s multi-step 1-of-N “ask for all transaction outputs, then ask for all transactions spending those outputs, and take the remainder” light-client model.

However, this state tree mechanism has an important disadvantage if implemented naively: the intermediate nodes in the tree greatly increase the amount of disk space required to store all the data. To see why, consider this diagram here:



The change in the tree during each individual block is fairly small, and the magic of the tree as a data structure is that most of the data can simply be referenced twice without being copied. However, even still, for every change to the state that is made, a logarithmically large number of nodes (ie. ~5 at 1000 nodes, ~10 at 1000000 nodes, ~15 at 1000000000 nodes) need to be stored twice, one version for the old tree and one version for the new trie. Eventually, as a node processes every block, we can thus expect the total disk space utilization to be, in computer science terms, roughly O(n\*log(n)), where n is the transaction load. In practical terms, the Ethereum blockchain is only 1.3 gigabytes, but the size of the database including all these extra nodes is 10-40 gigabytes.

So, what can we do? One backward-looking fix is to simply go ahead and implement headers-first syncing, essentially resetting new users’ hard disk consumption to zero, and allowing users to keep their hard disk consumption low by re-syncing every one or two months, but that is a somewhat ugly solution. The alternative approach is to implement ***state tree pruning***: essentially, use [**reference counting**](https://en.wikipedia.org/wiki/Reference_counting) to track when nodes in the tree (here using “node” in the computer-science term meaning “piece of data that is somewhere in a graph or tree structure”, not “computer on the network”) drop out of the tree, and at that point put them on “death row”: unless the node somehow becomes used again within the next X blocks (eg. X = 5000), after that number of blocks pass the node should be permanently deleted from the database. Essentially, we store the tree nodes that are part of the current state, and we even store recent history, but we do not store history older than 5000 blocks.

X should be set as low as possible to conserve space, but setting X too low compromises robustness: once this technique is implemented, a node cannot revert back more than X blocks without essentially completely restarting synchronization. Now, let’s see how this approach can be implemented fully, taking into account all of the corner cases:

1. When processing a block with number N, keep track of all nodes (in the state, tree and receipt trees) whose reference count drops to zero. Place the hashes of these nodes into a “death row” database in some kind of data structure so that the list can later be recalled by block number (specifically, block number N + X), and mark the node database entry itself as being deletion-worthy at block N + X.
2. If a node that is on death row gets re-instated (a practical example of this is account A acquiring some particular balance/nonce/code/storage combination f, then switching to a different value g, and then account B acquiring state f while the node for f is on death row), then increase its reference count back to one. If that node is deleted again at some future block M (with M > N), then put it back on the future block’s death row to be deleted at block M + X.
3. When you get to processing block N + X, recall the list of hashes that you logged back during block N. Check the node associated with each hash; if the node is still marked for deletion ***during that specific block*** (ie. not reinstated, and importantly not reinstated and then re-marked for deletion ***later***), delete it. Delete the list of hashes in the death row database as well.
4. Sometimes, the new head of a chain will not be on top of the previous head and you will need to revert a block. For these cases, you will need to keep in the database a journal of all changes to reference counts (that’s “journal” as in [**journaling file systems**](https://en.wikipedia.org/wiki/Journaling_file_system); essentially an ordered list of the changes made); when reverting a block, delete the death row list generated when producing that block, and undo the changes made according to the journal (and delete the journal when you’re done).
5. When processing a block, delete the journal at block N - X; you are not capable of reverting more than X blocks anyway, so the journal is superfluous (and, if kept, would in fact defeat the whole point of pruning).

Once this is done, the database should only be storing state nodes associated with the last Xblocks, so you will still have all the information you need from those blocks but nothing more. On top of this, there are further optimizations. Particularly, after X blocks, transaction and receipt trees should be deleted entirely, and even blocks may arguably be deleted as well – although there is an important argument for keeping some subset of “archive nodes” that store absolutely everything so as to help the rest of the network acquire the data that it needs.

Now, how much savings can this give us? As it turns out, quite a lot! Particularly, if we were to take the ultimate daredevil route and go X = 0 (ie. lose absolutely all ability to handle even single-block forks, storing no history whatsoever), then the size of the database would essentially be the size of the state: a value which, even now (this data was grabbed at block 670000) stands at roughly 40 megabytes – the majority of which is made up of [**accounts like this one**](https://explorer.etherapps.info/address/0x798d86e782c8c34da97f7389a464c8af76ea3442) with storage slots filled to deliberately spam the network. At X = 100000, we would get essentially the current size of 10-40 gigabytes, as most of the growth happened in the last hundred thousand blocks, and the extra space required for storing journals and death row lists would make up the rest of the difference. At every value in between, we can expect the disk space growth to be linear (ie. X = 10000 would take us about ninety percent of the way there to near-zero).

Note that we may want to pursue a hybrid strategy: keeping every ***block*** but not every ***state tree node***; in this case, we would need to add roughly 1.4 gigabytes to store the block data. It’s important to note that the cause of the blockchain size is NOT fast block times; currently, the block headers of the last three months make up roughly 300 megabytes, and the rest is transactions of the last one month, so at high levels of usage we can expect to continue to see transactions dominate. That said, light clients will also need to prune block headers if they are to survive in low-memory circumstances.

The strategy described above has been implemented in a very early alpha form in **[pyeth](https://github.com/ethereum/pyethereum/tree/pruning)**; it will be implemented properly in all clients in due time after Frontier launches, as such storage bloat is only a medium-term and not a short-term scalability concern.