Decentralized Storage on Large Dynamic Datasets with Applications for Large Decentralized KV Store

Qi Zhou, v0.1.6 draft

Oct. 3, 2022

Abstract

Decentralized storage on large dynamic datasets is one key challenge for future blockchains with large-scale applications. In this paper, we propose a dynamic data sharding to address the problem, especially its sub-problems such as proof of publication, proof of storage, and proof of retrievability. We apply the dynamic data sharding to a decentralized KV store on top of an EVM-compatible blockchain, and our early analysis suggests that the cost of storing large values can be reduced to $\approx$ 1/1000x compared to the fully-replicated EVM-native KV store via SSTORE opcode while ensuring tens or hundreds of replicas of the values in the network.

Contents

1 Introduction

One of the key challenges of future blockchains is storing a large amount of dynamic data that far exceeds the capacity of a single node. The storage problem can be generally divided into the following sub-problems:

- **Proof of Publication**, which ensures that the data show up on the network initially and the nodes can choose to download and store the data of interest or just ignore the data;
- **Proof of Storage**, which ensures that the data is stored somewhere in the network and prevents losing the data;
- **Proof of Retrievability**, which ensures that anyone can retrieve the data in the presence of some malicious nodes that withhold the data.

In this paper, we propose a dynamic data sharding to achieve decentralized storage on large dynamic datasets with applications for large decentralized key-value (KV) store on top of an EVM-compatible blockchain. The KV store maintains two data structures

- **Metadata of the KV store**, which is maintained in the on-chain KV contract and is fully replicated to all the nodes in the network;
- **Values of the KV store**, which are partitioned into multiple fixed-size shards. Each node may host zero or multiple shards and claim the rewards of each shard by submitting proof of storage periodically. When the number of KV entries increases, the shards will be dynamically created to serve new values of KV entries.

The key to achieving decentralized storage on large dynamic datasets is to build an on-chain oracle of the estimate of the number of replicas of each shard and then to reward those nodes that prove the replication of the shard over time. When a node is launched, the node operator is free to choose the shards (or no shards) to host, most likely depending on the shards that will offer the best-expected rewards, i.e., shards with a lower number of replicas based on the oracle. The reward of each shard is distributed to the nodes that can prove the replication of the shard via proof of random access (PoRA).

The PoRA is a mining process that heavily relies on read IOs that are performed over the shard data. Similar to the proof of work mining, the KV store contract maintains a dynamic difficulty parameter for each shard and adjusts the parameter after accepting the submission result of the PoRA (a.k.a., a mini-block). Therefore, we could estimate the number of replicas of the shard by calculating the hash rate, i.e., read IO rate, vs the read IO rate of most mining-economical storage devices (e.g., 1TB NVME SSD).

When some nodes of the shard leave the network, the hash rate of the shard will decrease, and therefore new nodes are incentivized to join the shard in order to receive better rewards. This dynamical procedure guarantees the replication over time and thus prevents loss of the data, i.e., achieving proof of storage.

Further, we could adjust the token reward (paid by users) to guarantee some levels of replication. Given the assumption that most of the nodes that run the shard replicas are honest (in fact, 1-of-N is sufficient), then we could address the data withholding attack from malicious nodes and thus achieve proof of retrievability.

Our early economic analysis shows that with the dynamic data sharding, the cost of storing KVs with large values can be decreased to $\approx 1/100$x of the current fully-replicated storage model in EVM (via SSTORE opcode) while ensuring tens or hundreds of replicas based on the IO performance of the targeted storage device.

The rest of paper are organized as follows. Section 2 describes the semantics of the proposed KV store. Section 3.1 explains the data structures of the KV store. Section 4 illustrates how to achieve proof of storage on large dynamic datasets. Section 5 discusses some limitations and optimizations of the KV store. Section 6 lists the attack vectors, and Section 7 concludes the paper.

2 Semantics of the KV Store

Consider an EVM-compatible blockchain network that supports the following decentralized key-value operations in Solidity application-binary-interface (ABI):

- `put(bytes32 key, bytes memory value)`, which writes the key-value pair to the underlying decentralized KV store. The size of the value must be smaller than or equal to the maximum value size `MAX_VALUE_SIZE` defined by the KV store. The value of `MAX_VALUE_SIZE` per-KV-store could be much larger than the existing Ethereum storage model (32 bytes), which ranges from a few kilobytes to hundreds of kilobytes;
- `get(bytes32 key, uint256 off, uint256 len) returns (bytes memory)`, which returns the value of the key from range [`off`, `off+len`). If the key is not found, it will return an empty bytes.
- `remove(bytes32 key)`, which removes the key-value pair from the KV store.
- `verify(bytes32 key, bytes memory value) returns (bool)`, which returns whether the value matches the underlying stored value.

To interact the KV store with the operations, an account (either externally owned or contract account) can call the KV store contract deployed at a specific address (e.g., 0x333001) with supplied parameters.

Note that we are expecting to support a very large amount of key-value pairs in the store. Therefore,

- each node maintains the full metadata of the KV store, which provides necessary functionalities such as ownership management, the key to physical address translation, and the hashes of values for integration check. The details of the metadata will be explained in Sec. 3.1.
- the actual values are stored in a large byte array. Since the array may contain tens or hundreds of Terabytes or even Petabytes of data, the content of the array will be partitioned into multiple shards that are hosted by different nodes in the network. The details of the value data storage will be explained in Sec. 3.2.

3 Data Structure of the KV Store

3.1 Metadata of the KV Store

The metadata of the KV store maintains two mapping in the contract:

- mapping from `skey` → `physical_addr`, where
  - `skey = keccak256(msg.sender, key)` is the actual storage key of the KV pair. Note that, even though all accounts share the same storage key space, since `keccak256()` hash function is cryptographically collision-free, we can safely assume that the storage keys derived from different accounts will never collide. As the result, the mapping implements a simple data ownership management, where each account will control its KV entries with the keys in a non-overlapping sub-space.
  - `physical_addr` is a tuple of

    $$(\texttt{uint40}\,\texttt{kv\_idx},\ \texttt{uint24}\,\texttt{kv\_size}, \texttt{bytes24}\,\texttt{hash}), \tag{1}$$

    where `kv_idx` is a unique index of the key-value pair , `kv_size` is the size of the value, and `hash` is the hash of the value of the pair. The sum of the sizes of the tuple is 256 bits (32 bytes), which can be stored in a single storage slot of the contract.
- mapping from `kv_idx` → `skey`, which maintains the reserve lookup from the index of a KV pair to its storage key.

When the put operation is called, the contract will do the following

1. Calculate `skey = keccak256(msg.sender, key)` and find if the storage key already exists.
2. If the KV pair of the storage key does not exist, then let `kv_idx = number of KV entries`, otherwise `kv_idx = existing index of the KV entry`.
3. Calculate new `physical_addr` and update the mapping.

```
// Write a large value to KV store.
// If the KV pair exists, overrides it.
// Otherwise, will append the KV to the KV array.
function put(bytes32 key, bytes memory data) public {
    bytes32 skey = keccak256(abi.encode(msg.sender, key));
    PhyAddr memory paddr = kvMap[skey];
    if (paddr.hash == 0) {
        // Append case
        paddr.kvIdx = lastKvIdx;
        idxMap[paddr.kvIdx] = skey;
        lastKvIdx = lastKvIdx + 1;
    }
    paddr.kvSize = uint24(data.length);
    paddr.hash = bytes24(keccak256(data));
    kvMap[skey] = paddr;

    // (Optionally) Write the value to the underlying data file
}
```

Figure 1: Example Code of Updating Metadata in put()

4. (Optionally) Store the value at `kv_idx`'th entry of data file.

An example code of updating metadata of the KV store can be found in Fig. 1.

3.2 Storage Layout of Value Data

The values of all KV entries are stored in a large byte array, i.e., a large-size file. Given the index of the KV entry `kv_idx`, the value will be written at the following position of the file

$$[\texttt{kv\_idx} \times \texttt{MAX\_KV\_SIZE}, \texttt{kv\_idx} \times \texttt{MAX\_KV\_SIZE} + \texttt{length\_of\_value}) \qquad (2)$$

where the range $[\texttt{kv\_idx} \times \texttt{MAX\_KV\_SIZE}, (\texttt{kv\_idx} + 1) \times \texttt{MAX\_KV\_SIZE})$ is reserved for the value of the KV entry for future in-place modification.

Since the file can be potentially huge, the content of the file is partitioned into multiple physical shards. Each shard contains the bytes of the file in the range

$$[\texttt{shard\_id} \times \texttt{SHARD\_SIZE}, (\texttt{shard\_id} + 1) \times \texttt{SHARD\_SIZE}), \qquad (3)$$

where `shard_id` is the index of the shard, and `SHARD_SIZE` is the number of bytes per shard defined as

$$\texttt{SHARD\_SIZE} = \texttt{MAX\_VALUE\_SIZE} \times \texttt{KV\_ENTRIES\_PER\_SHARD}, \tag{4}$$

which is configured by the KV store and the value of SHARD_SIZE should be fit into the per-node capacity with a few Terabytes.

Therefore, given the index of a KV entry, kv_idx, the corresponding shard index of the entry is

$$\texttt{shard\_id} = \left\lfloor \frac{\texttt{kv\_idx}}{\texttt{KV\_ENTRIES\_PER\_SHARD}} \right\rfloor. \tag{5}$$

When a node is launched, the node operator will be incentivized to configure the preferred shards that the node will serve. When a put operation is called, the node can determine whether the value of the KV entry should be stored locally given kv_idx of the entry and Eq. (5). If the KV entry does not belong to the shards of the node, the node will just discard the value.

One key problem of the sharding model is how to guarantee that the data of each shard is replicated with sufficient numbers so that

- (**Proof of Storage**) Even if some nodes that hold the shard data leave the network permanently, new nodes or the existing nodes are incentivized to join the shard, download the data, and serve the data in the network. This prevents the loss of data in the network.
- (**Proof of Retrievability**) The replication number is high enough that there exists (in a high probability) some honest nodes for the shard that will altruistically share its shard data with other nodes in the presence of malicious nodes withholding the data.

In the following sections, we will discuss how to address the issues.

- Section 4.1 describes the fee model for storing the KV data.
- Section 4.2 describes how a node with the shard data can prove its replication and claim rewards.
- Section 4.3 describes the estimated replication factor given some parameters of the fee model and physical storage cost.

4 Proof of Storage on Large Dynamic Datasets

4.1 Fee Model for Storage

To store the data in the proposed decentralized KV store, the user has to pay the storage cost to the nodes that host the shard data and to ensure sufficient replication factor. Here, we implement a storage rental model with discounted payment flow model. At time T_p (in seconds), the upfront payment of a first `put()` is

$$
\begin{aligned}
p &= c \sum_{t=T_p}^{\infty} d^{t-T_0} && (6)\\
&= \frac{cd^{T_p-T_0}}{1-d} && (7)\\
&= xd^{T_p-T_0}, && (8)
\end{aligned}
$$

where T_0 is the genesis time of the KV store, d is the discount rate in seconds, which reflects the decreasing cost of storage over time vs payment token, c a constant to adjust base payment, and $x = c/(1-d)$ is the upfront payment at genesis time. When the user puts the same key and replace the value, we will not charge the storage rental cost again (normal gas fee is still charged).

The payment can be refunded if the KV entry is removed. Suppose the removal time is T_r, then the refund will be

$$
\begin{aligned}
r &= c \sum_{t=T_r}^{\infty} d^{t-T_0} && (9)\\
&= xd^{T_r-T_0}. && (10)
\end{aligned}
$$

As a result, for the time interval $[T_p, T_r]$, the user rents the storage and pays the rental fee as

$$
\begin{aligned}
f &= p - r && (11)\\
&= x(d^{T_p-T_0} - d^{T_r-T_0}), && (12)
\end{aligned}
$$

which can be computed efficiently in smart contracts.

4.2 Proof of Replication

To incentivize nodes to host the data of each shard, we employ a proof of replication algorithm based on proof of random access. The goal of proof of random access is to provide an on-chain oracle about the number of read IOs (in terms of `MIN_IO_SIZE`, e.g., 4096 for most SSDs) are performed over the shard data over time. For example, suppose that the oracle reports 10,000,000 4KB read IO/s and a normal NVME SSD can offer 500,000 4KB read IO/s, then we could estimate about 20 disks that are hosting the shard data in the network.

To estimate the read IO rate, each node hosting the shard data will perform a mining by randomly reading the shard data as follows.

1. Based on a nonce and the last mining hash, randomly pick up `N_RANDOM_ACCESS` positions in the range of

$$[0, \texttt{SHARD\_SIZE}/\texttt{MIN\_IO\_SIZE}) \tag{13}$$

2. Read the data from the shard at range

$$[p_i \times \texttt{MIN\_IO\_SIZE}, (p_i + 1) \times \texttt{MIN\_IO\_SIZE}), \tag{14}$$

where $p_i, i \in \{1, ..., \texttt{N\_RANDOM\_ACCESS}\}$ are the random positions generated from Step 1.

3. Mix the randomly read data (e.g., simply XOR the data read from random positions) and calculate the final hash H

4. Check if

$$\left\lfloor \frac{2^{256} - 1}{H} \right\rfloor \geq D_i \tag{15}$$

where D_i is the difficulty of the shard.

5. If Eq. (15) is satisfied, submit the mining result as an EVM transaction to the KV store contract, which will verify the result with on-chain metadata and send the rewards over $[T_{prev}, T]$ following Eq. (11) as

$$x(d^{T_{prev} - T_0} - d^{T - T_0})n_i, \tag{16}$$

where T_{prev} is the timestamp of the previous submission, T is the current timestamp, and n_i is the number of the KV entries paid in the shard i. The difficulty of the shard D_i is automatically adjusted by the KV store contract given the submission internal $T - T_{prev}$ and target submission interval T_{target} similar to Ethash difficulty adjustment algorithm.

6. Repeat Step 1.

As the result, the IO rate of the shard in the network can be estimated as

$$\frac{D_i \times \texttt{N\_RANDOM\_ACCESS}}{T_{target}}, \tag{17}$$

which can be served as an on-chain oracle.

4.3 Estimated Replicas

In this subsection, we study the estimated number of physical replicas. Firstly, the cost of hosting and mining 1TB data is analyzed as follows.

- $\texttt{MAX\_VALUE\_SIZE} = 4096\ \texttt{bytes}$
- $\texttt{SHARD\_SIZE} = \texttt{1TB} = 1024^4\ \texttt{bytes}$
- Cost of storage is \$100 (e.g., Samsung 970 EVO Plus with 1TB), amortized over 4 years with yearly cost = \$25
- Yearly power cost = $6\texttt{W} \times 24 \times 365 = 52.56\texttt{kWh} \approx \$26.28$ @ \$0.5 per kWh
- Total yearly cost \$51.28

The reward distributed to the nodes that host the replicas of the shard depends on upfront payment x, discount rate d, and token price. For simplicity, we use a ETH-like token with 18 decimals (in the unit of Wei), and analyze the first year reward with the following parameters

- $x = 0.0001\ \texttt{TOKEN} = 10^{14}\texttt{Wei}$, i.e., per-byte storage cost is $10^{14}/4096 = 24.4\texttt{Gwei}$. As a comparison, the current Ethereum storage cost per-byte via SSTORE opcode is

  $$20000/32 \times gas\_price, \tag{18}$$

  which means that SSTORE will have the same or lower storage cost if $gas\_price \leq 0.039\texttt{Gwei}$, which is far lower than current average gas price of Ethereum mainnet, which is around 20Gwei.
- Yearly discount $= d^{365\times 24\times 3600} = 0.9$
- First year token rewards $= (1 - yearly\_discount) \times 0.0001 \times 1024^4/4096 = 2684.35456$ tokens
- Profit margin $= 50\%$

As a result, the estimated replicas of the first year can be obtained as

$$replicas = \frac{first\_year\_rewards \times token\_price \times (1 - profit\_margin)}{yearly\_cost} \tag{19}$$

Table 1 summarizes the estimated replicas given different values of token prices.

Token price	\$1	\$5	\$10	\$100
Estimate replicas	26.8	134.2	268	2680

Table 1: Estimated Replicas of a Shard.

5 Other Limitations and Optimizations

5.1 Dynamically Creating a New Shard

When the number of KV entries N is increased from $i \times \texttt{KV\_ENTRIES\_PER\_SHARD}$ to $i \times \texttt{KV\_ENTRIES\_PER\_SHARD} + 1$, a new shard with id $i+1$ is dynamically created. One issue with the new shard is that it may not have any node that is configured to serve this shard at the time of creation due to the lack of incentive, i.e., no mining reward just before its creation. This may result in losing the value of a newly put KV entry in the new shard.

To address this issue, one solution is to pre-pay some amount of the KV entries in the KV store contract, N_PREPAID, such that the actual number of the KV entries paid in the store is $N + \texttt{N\_PREPAID}$. This provides a grace period to incentivize the nodes to join the shard to be created and mine the data (although the shard may not contain any values, we will fill it with some random values so that proof of random access can still work, See Section 6.3). As the result, the number of paid KV entries in shard i, i.e., n_i, $(i \geq 0)$ in Eq. 16 becomes

$$n_i = \min\left(\texttt{KV\_ENTRIES\_PER\_SHARD}, (N + \texttt{N\_PREPAID}) - i \times \texttt{KV\_ENTRIES\_PER\_SHARD}\right) \tag{20}$$

5.2 Limitations on get()

Since the values of the KV entries are sharded in different nodes, a consensus node may not be able to serve a get operation if it is called in a transaction of a block. Because of this, we do not allow the get operation to be called in the consensus execution, e.g., proposing a block or validating a block. However, a transaction can include the value of the KV entry and verify it on-chain using `verify()` method.

In addition, a contract can still call the get operation in the JSON-RPC environment, e.g., via `eth_call` JSON-RPC method. The input argument of a JSON-RPC call may further include a list of values that the contract may read. During the contract execution, if the values are not found in the argument, the call will try to find the value data locally. If the local data file does not contain the data either, the call will fail and return the error containing the expected value (and its key and index) to supply. This allows the caller to find the values from another node and repeat the process until the call is successful.

5.3 Limitations on remove()

Removing a KV entry generally requires moving the last value of the data file to replace the value of the KV entry to be removed. The example code can be found in Fig. 2.

One key issue is that the removal may perform a cross-shard data moving, i.e., moving the value data from the last index of the value of the data file to the index of the KV to be removed, where the values are in different shards. Since a node may not serve both shards, the move may result in the loss of data if no node serves both shards. To address this issue, we can have the following methods:

- We only allow removing a KV entry in the last shard i, i.e.,

  $$i = \lfloor (N + \texttt{N\_PREPAID}) / \texttt{KV\_ENTRIES\_PER\_SHARD} \rfloor \tag{21}$$

  and thus moving the data always happens in the same shard; or
- Removing cross-shard data must supply the value to be moved; or
- Appending the KV index in a free list in the contract and refunding the users only when a `put()` operation removes the KV index from the free list.

```
// Remove an existing KV pair to a recipient.
// Refund the remaining payment accordingly.
function remove(bytes32 key) public {
    bytes32 skey = keccak256(abi.encode(msg.sender, key));
    PhyAddr memory paddr = kvMap[skey];
    uint40 kvIdx = paddr.kvIdx;

    require(paddr.hash != 0, "KV not exist");

    // Move last KV metadata to current KV
    bytes32 lastSkey = idxMap[lastKvIdx - 1];
    idxMap[kvIdx] = lastSkey;
    kvMap[lastSkey].kvIdx = kvIdx;

    // Clear the metadata of the removing KV
    kvMap[skey] = PhyAddr({kvIdx: 0, kvSize: 0, hash: 0});

    // Remove the last KV metadata
    idxMap[lastKvIdx - 1] = 0x0;
    lastKvIdx = lastKvIdx - 1;
    // (Optional) Move the lastKvIdx'th data to kvIdx'th
    // in local data file.

    // Refund owner.
}
```

Figure 2: Example Code of Removing Metadata in remove()

5.4 Storage Capacity of the Network

The current storage capacity of the network is primarily limited by the capacity of storing the metadata, which will be fully replicated to all the nodes. Each KV entry will consume 2 contract storage slots with $2 \times (32 + 32) = 128$ bytes. We further assume the overhead of the Partica Merkle Tree (MPT) of the contract storage is about 2x, and thus the overhead of metadata is about 256 bytes per KV entry. Given the capacity of a node is about 2TB and different `MAX_VALUE_SIZE`'s, we could estimate the storage capacity of the network in Table 2.

`MAX_VALUE_SIZE`	4KB	16KB	64KB	256KB
Storage Capacity	32TB	128TB	512TB	2PB

Table 2: Network Storage Capacity with Different `MAX_VALUE_SIZE`'s.

Note that we may further increase the capacity by sharding the metadata together with the KV data. Therefore, instead of storing all metadata on-chain, we only need to maintain the Merkle root of the metadata/data for each shard in the smart contracts. This will save a huge amount of fully-replicated data at the cost of complicating the semantics of all KV operations (e.g., a put operation will require additional Merkle proof).

5.5 Optimization on Uploading

Current uploading off-chain data will place the data in the calldata field of a transaction. Considering the gas cost per byte in calldata is about 16, uploading a large object will consume a significant amount of gas, e.g., a 256KB object will take about 4 million gas.

One optimization of uploading is to use EIP-4488: Transaction calldata gas cost reduction with total calldata limit. This will reduce the gas cost per byte in calldata to 3, while setting the hard limit of block size to prevent the worst-case attack.

A further optimization can use BlockShadow as proposed by Arweave and compact blocks in BIP-152. The basic idea is that when announcing a new block, the node only propagates a compact block that only contains the transaction hashes. A node receiving the compact block can reconstruct the full block by filling the transactions found in local memory pool. If some of the transactions are missing in the pool, the receiving node will ask the source node for the transactions with extra delays. Assuming that the nodes (especially the validators) maintain the latest pending transactions in its memory pool, reconstructing the full block from a compact block can be done with a high probability. As a result, we may further increase the block size limit and decrease gas cost per byte in calldata after implementing EIP-4488.

A long-term optimization is to employ the latest data availability (DA) technologies. For example, with danksharding, the upload speed of future Ethereum will be about 2.66 MBps, which is about 20x of current Ethereum upload speed (about . The benefit of DA is that the nodes do not need to download and broadcast all the calldata - instead, every node can verify that any part of data can be downloaded from the network via data availability sampling. This model fits very well in our dynamic sharding model, where the nodes only need to download the data of the shards of interest without download all of them.

5.6 Optimization on the Operational Cost of a Node

Current implementation based on Geth will store all historical blocks in the local storage. Since uploading the data will put the data in the transactions, maintaining all historical blocks can easily exceed per-node storage capacity as the size of uploaded data grows. To bound the storage size of a consensus node, EIP-4444: Bound Historical Data in Execution Clients can be implemented to prune the historical blocks and to only maintain recent blocks. Assuming the upload rate of the network is 2MB/s, limiting the historical block data to 1T will ask the node to keep recent blocks of

$$1024^4/2/1024^2/24/3600 \approx 6\ \texttt{days}. \tag{22}$$

5.7 Synchronizing Value Data

When a fresh node is launched with configured shards to serve, the node needs to synchronize the values of the shards before the node could prove its replication via PoRA and claim rewards. To support efficient synchronization, we will develop a devp2p sub-protocol to discover the nodes that serve the shard data and synchronize the data. The synchronization part is similar to snapsync subprotocol in devp2p, where values are copied from the remote peers concurrently to maximize the bandwidth and IOs.

6 Attack Vectors

6.1 Overestimate of Replicas with Partial Replicas

If a node stores partial data of a shard, it may still achieve the full hash rate by skipping the random access of unstored data in Step 1 in Section 4.2. Therefore, the full replicas in the network may be over-estimated since some nodes may not maintain full replicas of the shard while mining at the full IO rate. To encourage the nodes to fully synchronize the replica, we can adopt a Hashimoto-like mining procedure:

- The random position p_i depends on the data of previous position, i.e., the data from the shard at range $[p_{i-1} \times \texttt{MIN\_IO\_SIZE}, (p_{i-1}+1) \times \texttt{MIN\_IO\_SIZE})$;
- We could further increase the value of N_RANDOM_ACCESS, which will decrease the success rate of PoRA exponentially. E.g., suppose N_RANDOM_ACCESS =

16 and the node only stores 90% of the data of the shard, then the success rate of mining is

$$0.9^{16} \approx 0.1853 \tag{23}$$

6.2 Overestimate of Replicas with Advanced Mining Machine

Another possible vulnerability of PoRA is the potential overestimation of the replicas if mining machines with high IO bandwidth are employed. One example of such a high bandwidth device is memory, which could be much faster than SSDs. However, compared to 1TB NVME SSD (Samsung 970 EVO Plus) with about $100 cost, the memory cost of 1TB is much higher (about 40x, e.g., 256GB DDR4-3200 is about $1000, and we could safely assume 1TB memory is about $4000). The copy bandwidth of such memory (measured in 1 byte read and write) is about 12.8 GB/s, which is about 6x of 1TB NVME SSD at 2GB/s (Samsung 970 EVO Plus with up to 550K 4KB random reads with queue depth 32). As a result, using memory seems to be much less economically efficient compared to NVME SSDs. The efficiency gap can be further increased by increasing the size of the shard `SHARD_SIZE`.

6.3 Unfair Mining Advantage with Attacker Generated Data

If parts of the shard data are generated by an attacker (e.g., from a fast pseudo-random number generator with a secret seed only known to the attacker), the attacker can achieve a mining advantage when the attacker's data is accessed in PoRA. The attacker can generate the data in-flight without performing actual IOs. If the percentage of the attacker's hash power over total hash power of the shard is greater than the percentage of the attacker's generated data over the shard data, then the attacker will have an unfair mining advantage over other honest miners.

A way to address the problem is to adopt Dagger-like mining, where the actual data stored on disk is

$$v_i \oplus \mathtt{DAG}_i \tag{24}$$

where v_i is the value of the ith KV entry, $\oplus$ is a bit-wise xor operator, and $\mathtt{DAG}_i$ is the ith item of a directed acyclic graph (DAG) data set. Similar to Ethash, the DAG data set can be generated from a smaller cache, which allows light verification for non-mining nodes.

6.4 Front-Running Attack

When a miner submits a transaction with a PoRA result to the contract, an attacker can immediately detect the access positions of the submission when the transaction is broadcast. If the attacker owns the KV entry corresponding to one of the positions, then the attacker can invalidate the submission by modifying the value of the KV entry before the transaction is included in the blockchain.

One way to address the attack is to allow a PoRA submission to use the metadata of a recent block. Since the current Geth implementation maintains the states of the recent blocks, reading the metadata in a recent block's state should have almost the cost as reading the metadata in the current state. This gives a grace period that the miner can mine a recent snapshot of the values in a shard until the snapshot expires.

7 Conclusion

In this paper, we proposed the decentralized storage solution for a large amount of dynamic data with applications for a decentralized KV store. The proposed solution fully replicas the KV store metadata on a smart contract of an EVM-compatible blockchain, while partitioning the values of all KV entries into multiple shards. Each shard is incentivized to be replicated on multiple physical disks to ensure redundancy. Our early estimation shows that the network capacity of the proposed solution can achieve tens or hundreds of Terabytes or even Petabytes.