GEM^2-Tree: A Gas-Efficient Structure for Authenticated Range Queries in Blockchain

Ce Zhang*, Cheng Xu*, Jianliang Xu*, Yuzhe Tang‡, Byron Choi*

* *Department of Computer Science, Hong Kong Baptist University, Hong Kong*
‡ *Department of Electrical Engineering & Computer Science, Syracuse University, NY, USA*
{cezhang, chengxu, xujl, bchoi}@comp.hkbu.edu.hk, ytang100@syr.edu

***Abstract*—Blockchain technology has attracted much attention due to the great success of the cryptocurrencies. Owing to its immutability property and consensus protocol, blockchain offers a new solution for trusted storage and computation services. To scale up the services, prior research has suggested a hybrid storage architecture, where only small meta-data are stored on-chain and the raw data are outsourced to off-chain storage. To protect data integrity, a cryptographic proof can be constructed online for queries over the data stored in the system. However, the previous schemes only support simple key-value queries. In this paper, we take the first step toward studying authenticated range queries in the hybrid-storage blockchain. The key challenge lies in how to design an authenticated data structure (ADS) that can be efficiently maintained by the blockchain, in which a unique *gas* cost model is employed. By analyzing the performance of the existing techniques, we propose a novel ADS, called GEM^2-tree, which is not only gas-efficient but also effective in supporting authenticated queries. To further reduce the ADS maintenance cost without sacrificing much the query performance, we also propose an optimized structure, GEM^{2*}-tree, by designing a two-level index structure. Theoretical analysis and empirical evaluation validate the performance of the proposed ADSs.**

I. Introduction

Blockchain technology has been receiving unprecedented attention thanks to its wide applications in various fields, such as finance, healthcare, IoT, and supply chain management [1], [2]. The blockchain is a secure data structure which can be maintained by untrusted peers in a decentralized P2P network. The integrity of the data stored in the blockchain is upheld through two security designs: the hash-chain technique and the consensus protocol [3]. They together ensure that the data stored in the blockchain are immutable and that each peer in the network stores the same replicas of the data.

The blockchain was originally invented to serve as a transaction ledger for the cryptocurrency Bitcoin [4]. More recently, with the emergence of the second-generation blockchain represented by Ethereum [5], the technology has also been adopted as a trusted storage solution for more general data, such as text, documents, and images [6]. Since such general data are usually large, storing raw data directly on-chain is not scalable. To tackle this issue, a common approach is to employ a hybrid storage architecture [7], [8]. As shown in Fig. 1, the data owners (e.g., IoT devices) continuously send the data to the blockchain for secure storage. The raw data is stored off-chain on a dedicated storage server (e.g., Amazon S3 or Google Cloud Storage). At the same time, a cryptographic hash of each data object is kept on-chain as notarization of the raw data. To ensure data integrity, on-chain hashes are used to authenticate the data retrieved from the off-chain storage. While this scheme works well for simple key-value queries, range queries, another widely used query type, are not supported.

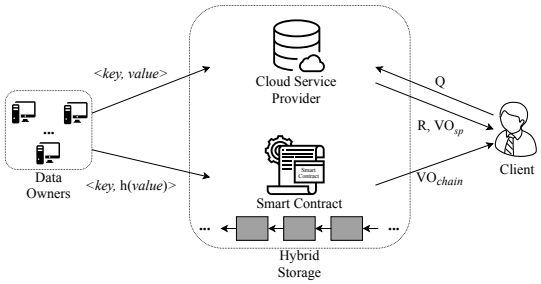


Fig. 1. Authenticated Query Framework in Hybrid-Storage Blockchain

In this paper, we take the first step toward studying authenticated range queries in the hybrid-storage blockchain. Inspired by authenticated query processing in outsourced databases [9], an intuitive approach is to leverage the *smart contract*[1] to construct an authenticated data structure (ADS, e.g., Merkle hash tree [10]) on top of the search keys in the blockchain. Meanwhile, a similar ADS is maintained by the cloud service provider. Based on the ADS, a verification object (VO) can be generated for each query and returned along with the result. Using the VO, the client is able to verify whether or not the query result is both *sound* and *complete*. Here *soundness* means that all of the answers satisfy the query condition and truly originate from the data owners, and *completeness* means that no valid answer is missing.

The major challenge of the above approach comes from data updates. To keep track of the updates, the ADS needs to be dynamically maintained by the smart contract. In a smart contract-enabled blockchain like Ethereum, users need to pay *gas* for storage and computation as the smart contract execution costs *miner*'s resources. The amount of gas to pay for different operations differs. Notably, the gas charged for a smart contract write operation is several orders of magnitude

[1]The smart contract is a trusted program running on the top of the blockchain, whose execution integrity is ensured by the consensus protocol of the blockchain.

higher than that for a read operation (e.g., 20,000 vs. 200 in Ethereum). Thus, if we simply maintain a full Merkle tree as the ADS, the update cost would be prohibitively high. The reason is threefold: (i) an insertion may incur a series of updates in the leaf node to preserve the order of the data; (ii) an insertion entails updating the hashes of all ancestor nodes; (iii) an insertion may lead to recursive node splits, which consume lots of storage and computation for the creation of new nodes and redistribution of index keys. As such, we need to design novel ADSs that allow efficient updates in terms of the gas cost.

To this end, we propose a new ADS, called Gas-Efficient Merkle Merge Tree (GEM^2-tree), that can be efficiently maintained in the blockchain while being effective in supporting authenticated range queries. The main idea of the GEM^2-tree is to trade writes for reads and computations. On the one hand, we do not maintain a single full-tree structure in the blockchain, but multiple partial trees that can be gracefully merged with more objects inserted. This helps to reduce the update costs, although more reads will be incurred for query authentication. On the other hand, some internal nodes of the GEM^2-tree are suppressed and computed on the fly to maintain the root hashes, which are needed for result verification. In this way, update costs are reduced at the expense of more computations. To further reduce the ADS maintenance cost, we propose an optimized ADS, called GEM^{2*}-tree. This extends the GEM^2-tree with an upper-level index that splits the search key domain into several non-overlapping subspaces.

To summarize, this paper's contributions are as follows:

- For the first time in the literature, we study the problem of authenticated range queries in the hybrid-storage blockchain.
- We propose a gas-efficient ADS, called GEM^2-tree, that can significantly reduce the storage and computation costs of the smart contract.
- We develop an optimized ADS, GEM^{2*}-tree, which can further reduce the maintenance cost without sacrificing much the query performance.
- We conduct theoretical analysis and empirical evaluation to validate the performance of the proposed ADSs. Experimental results show that our proposed ADSs, in comparison with the traditional methods, can reduce the gas cost by a factor of up to 4 with little penalty on the query performance.

The rest of the paper is organized as follows. Section II introduces some preliminaries, followed by the problem formulation in Section III. Section IV discusses the baseline solutions. Our solution is presented in Section V, which is then further optimized in Section VI. Section VII presents the experimental results. Finally, Section VIII reviews related studies, and the paper is concluded in Section IX.

II. Preliminaries

In this section, we give some preliminaries that will be used in the subsequent sections.

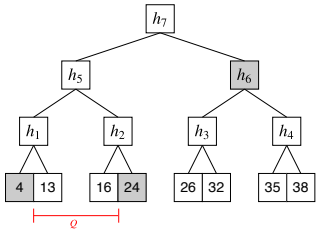


Fig. 2. Merkle Hash Tree

A. Cryptography Primitives

Cryptographic Hash Function: A cryptographic hash function $\mathsf{h}(\cdot)$ maps an arbitrary-length message m to a fixed-length message digest $\mathsf{h}(m)$. It has two important properties: one-way and collision resistance. The one-way property indicates that given a digest $\mathsf{h}(m)$, a PPT adversary can find the original message m with a negligible probability. On the other hand, collision resistance means that it is computationally infeasible for a PPT adversary to find two different messages m_1 and m_2 such that $\mathsf{h}(m_1) = \mathsf{h}(m_2)$.

Merkle Hash Tree [10]: A Merkle Hash Tree (MHT) is a data structure that can be used to authenticate a set of data objects with logarithmic time complexity. It is widely used in authenticated queries and also in the blockchain structure. Fig. 2 shows an example of an MHT with eight data objects. Generally, the MHT is a binary tree constructed bottom-up. Each leaf node contains the hashes of the indexed objects.[2] Each internal node contains a hash which is computed using its two child nodes (e.g., $h_5 = \mathsf{h}(h_1||h_2)$, where "||" denotes string concatenation). Owing to the collision resistance property of the hash function, the root hash (i.e., h_7 in Fig. 2) can be used to authenticate the data objects stored in the leaf nodes. For example, if a range query $Q = [10, 20]$ is asked, the result is $\{13, 16\}$, and one can construct a proof consisting of $\{4, 24, h_6\}$ (shaded part in Fig. 2). A verifier can reconstruct the root hash using the result and proof, and further compare it with the signed root hash, which is publicly available. If they match, it means the result has not been tampered with. Furthermore, the boundary objects 4 and 24 guarantee the completeness of the result.

The MHT concept has been extended to various database indexes to suit different query applications. The Merkle B-tree (or MB-tree) [9] is one of such examples, which combines B-tree and MHT to support authenticated queries for outsourced relational databases. While the structure of MB-tree is based on the traditional B-tree, like MHT, each index entry of MB-tree is augmented with a corresponding hash. MB-tree can be seen as a generalized MHT in which the fanout of the tree is increased from binary to m-ary.

B. Blockchain and Smart Contract

A blockchain consists of a series of blocks chained by cryptographic hash pointers (see Fig. 3). Each block stores a list of transaction records and an MHT is built on top of

[2]In Fig. 2, for clarity, we simply use the search key value to denote the hash of an object.

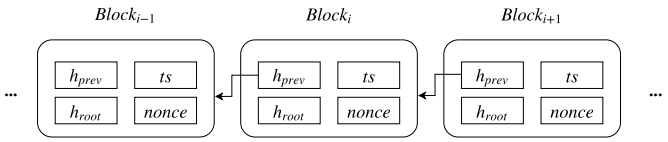


Fig. 3. Blockchain Structure

TABLE I
ETHEREUM GAS COST (A FULL LIST IS AVAILABLE IN [5])

Operation	Gas Used	Explanation
C_{sload}	200	load a word from storage
C_{sstore}	20,000	store a word to storage
$C_{supdate}$	5,000	update a word to storage
C_{mem}	3	access a word in memory
C_{hash}	$30 + 6 \cdot \lvert words \rvert$	hash an arbitrary-length data

the transaction records. The header of a block contains a cryptographic hash of the previous block h_{prev}, a timestamp ts, an MHT root hash h_{root}, and a consensus-proof $nonce$ that is found by consensus peers (known as *miners*). To append a new block to the blockchain, a miner needs to find a qualified $nonce$ and broadcast it to the entire network. In the Proof of Work (PoW) consensus protocol [4], $nonce$ should satisfy $\mathsf{h}(ts|h_{prev}|h_{root}|nonce) < D$, where D is a small value used to control the difficulty level of the *mining* process. Upon receiving a new block, other miners verify the hashes and the $nonce$ and, once verified, add the new block to the blockchain. The blockchain protocol ensures that each peer keeps the same replicas of the data and the stored data are immutable.

A smart contract is a trusted program that allows users to process data in the blockchain. The program is executed by the miners and its correctness is guaranteed by the blockchain consensus protocol. A deployed contract can be triggered by the transactions recorded in the blockchain. During execution, a transaction fee, denominated in *gas*, is charged as the miners spend computational resources. Table I shows the fees for some major storage and computation operations in the Ethereum platform [5]. As can be seen, the operation of storing data to the blockchain is more expensive than that of updating data, which is itself more costly than reading data from the blockchain and the in-memory operations. Furthermore, to prevent a smart contract from wasting too many computation resources of the miners, a *gasLimit* (e.g., 8,000,000 in [5]) is introduced. If the total gas consumption exceeds the *gasLimit*, the execution will be aborted. As such, it is of the utmost importance to minimize the gas consumption.

III. PROBLEM FORMULATION

A. System Model

As shown in Fig. 1, our system consists of four parties: data owners (DO), a blockchain with smart contract functionality, a cloud service provider (SP), and query clients. The blockchain itself and the SP are components of the hybrid-storage blockchain. Each data object is modeled as a tuple $o_i = \langle k_i, v_i \rangle$, where k_i is the value of the search key and v_i denotes the rest of the data object. During data insertions or updates, the DO sends $o_i = \langle k_i, v_i \rangle$ to the SP and sends $\langle k_i, \mathsf{h}(v_i) \rangle$ to the blockchain. Note that as the blockchain is used for query authentication, the hash value $\mathsf{h}(v_i)$, rather than v_i itself, is stored in the blockchain. This can help reduce the storage cost without compromising the guarantee of integrity.

To facilitate authenticated query processing and result verification, an *authenticated data structure* (ADS) should be maintained by both the SP and the smart contract of the blockchain. Upon receiving a data insertion or data update, the smart contract is triggered to update the ADS in the blockchain. Meanwhile, the ADS in the SP is updated accordingly. The digest of the ADS becomes *authenticated information* that is shared by both the SP and the smart contract.

In this paper, we mainly focus on range queries. The query processing procedure is as follows. The client sends a query to the SP, which uses the ADS to compute the query result as well as a *verification object* (VO_{sp}) that contains the information for the client to verify the result. Both the query result and the VO are returned to the client. During result verification, the client first retrieves the authenticated digest (hereafter denoted VO_{chain}) from the blockchain. Then, by combining the VO_{sp} from the SP and the VO_{chain} from the blockchain, the client can verify the correctness of the returned result.

B. Threat Model

In our model, the DO, the blockchain, and the query client are assumed to be trusted parties. The third-party SP is seen as an untrusted party since it may modify, add, or delete data intentionally or unintentionally [11]. Therefore, the SP is required to prove the *soundness* and *completeness* of the query result:

- **Soundness**. All of the answers in the result satisfy the query criteria and are originated from the DO;
- **Completeness**. No valid answer is missing from the query result.

The security notions will be formalized when we perform security analysis in Section V-E.

With the above system model and threat model, the problem we are going to study in this paper is how to design an ADS that can be efficiently maintained by the smart contract, in terms of the *gas* cost, while effectively supporting authenticated range queries. In the following sections, we first present two baseline solutions and then propose a novel gas-efficient ADS.

IV. BASELINE SOLUTIONS

In this section, we present two baselines solutions, namely Merkle B-tree (MB-tree) and Suppressed Merkle B-tree (SMB-tree). The general idea is that the SP and the blockchain both maintain a version of MB-tree to support authenticated queries over the hybrid-storage blockchain.

A. Merkle B-tree (MB-tree)

As introduced in Section II, the MB-tree can be used to authenticate range queries. Thus, intuitively, two identical MB-trees can be constructed and maintained as ADS by the SP and

the smart contract of the blockchain, respectively, except that the actual data objects are not stored in the blockchain. On the SP side, whenever there is a query from the client, the SP can traverse the MB-tree to construct a VO_{sp}. For example in Fig. 2, given a query $Q = [10, 20]$, $VO_{sp} = \{4, 24, h_6\}$. For result verification, the client first retrieves the authenticated digest $VO_{chain} = h_7$ from the blockchain. Then the MB-tree root is reconstructed locally using the result $\{13, 16\}$ and VO_{sp}. Since the MB-trees maintained by the blockchain and the SP are identical, the client can establish the soundness of the result by checking the reconstructed root hash against the one retrieved from the blockchain (i.e., VO_{chain}).

We next analyze the maintenance cost of the MB-tree in the blockchain. For the sake of simplicity, we consider the case of inserting a single object. To optimize the gas cost, we assume that the MB-tree's node capacity is the same as the granularity of blockchain data access. Suppose that the fanout of the MB-tree is F and the current database size is N. First, an object insertion requires finding the leaf node to store the object, which consumes $\log_F N \cdot C_{sload}$ gas. The inserted object costs an additional C_{sstore} gas. Second, an object insertion demands hash updating of $\log_F N$ ancestor nodes, each one requiring $F \cdot C_{sload} + C_{hash} + C_{supdate}$ gas. Furthermore, in the worst case, an object insertion could result in $O(\log_F N)$ node splits to maintain balanced tree structure. In each node split, a new node will be created along with the key redistribution and the updating of the nodes' hash values. A node creation consumes $2C_{sstore}$ gas for storage of the node's content and hash, whereas the rest of the operations contribute to $F \cdot C_{sload} + C_{supdate}$ gas consumption. In total, a single object will yield the following gas cost in the worst case:

$$C^{\text{insert}}_{\text{MB-tree}} = \log_F N \Big(2C_{sstore} + 2C_{supdate} + (2F+1)C_{sload} + C_{hash} \Big) + C_{sstore}$$

It can be observed that the cost increases logarithmically with the database size N. It is also worth noting that among all smart contract operations, C_{sstore} and $C_{supdate}$ are more expensive than the others (see Table I).

B. Suppressed Merkle B-tree (SMB-tree)

The maintenance of the MB-tree in the blockchain would incur a large amount of gas consumption due to the extensive write operations (i.e., $sstore$ and $supdate$). At the same time, it can be observed that only the root hash VO_{chain} is used during the query processing. Therefore, an alternative solution is to suppress all nodes of the MB-tree and only materialize the root node in the blockchain. We call this structure the *Suppressed Merkle B-tree* (SMB-tree). During each object insertion, the smart contract will compute all nodes of the SMB-tree on the fly and only update the root hash to the blockchain storage. Note that the MB-tree in the SP is maintained in the same way but not suppressed.

Similar to the MB-tree, we analyze the gas cost for a single object insertion. The first step of the smart contract is to load all data into the memory from the blockchain storage. This step incurs $N \cdot C_{sload}$ gas consumption. Next, the loaded objects are sorted, which requires $N \log N \cdot C_{mem}$ gas. Once the objects are sorted, the smart contract can compute all the MB-tree hashes on the fly with $N/F \cdot C_{hash}$ gas. Finally, the inserted object and the updated root hash are written into the blockchain storage, which incurs an additional $C_{sstore} + C_{supdate}$ cost. In total, the SMB-tree involves the following gas cost for each object insertion:

$$C^{\text{insert}}_{\text{SMB-tree}} = N \Big(C_{sload} + \log N \cdot C_{mem} + \frac{1}{F} C_{hash} \Big) + C_{sstore} + C_{supdate}$$

Compared with the normal MB-tree, the SMB-tree yields a gas cost in the complexity of $O(N \log N)$ with respect to the database size. Nevertheless, because the read operation (i.e., $sload$) and the in-memory operations (e.g., mem and $hash$) are several orders of magnitude cheaper than the write operations, the SMB-tree has the potential to reduce gas consumption for a small to medium N. On the other hand, $C_{\text{SMB-tree}}$ will surpass $C_{\text{MB-tree}}$ with a sufficiently large N.

C. ADS Design Principles

Based on the cost analysis of the baseline solutions, we consider the following principles to design an optimized ADS which is efficient in both maintenance and query authentication.

- *Avoid maintaining long sorted lists.* The insertion of an N-length sorted list costs $N/2 \cdot C_{supdate}$ gas on average. The high update cost will weaken the performance when database size increases.
- *Use more reads instead of writes.* The write cost in the blockchain is much higher than the read cost due to the consensus protocol. Thus, for intermediate variables, we may compute them in the memory and maintain only the final computation result in the blockchain to reduce the storage cost.
- *Be adaptable to databases of different sizes.* The database size has an impact on the maintenance performance of an ADS. An ideal ADS should be able to adapt itself to the database size.

V. Gas-Efficient Merkle Merge Tree

Following the above design principles, in this section we propose a new ADS, called Gas-Efficient Merkle Merge Tree (GEM^2-tree). The GEM^2-tree not only can be maintained by the smart contract with optimized gas performance, but is also capable to support authenticated queries efficiently.

A. GEM^2-tree Structure

As discussed in Section IV, the MB-tree and SMB-tree are efficient for large databases and small databases, respectively. Thus, in the GEM^2-tree, we maintain multiple separate structures: a large fully-structured MB-tree as the major index and a series of small structure-suppressed SMB-trees to index newly

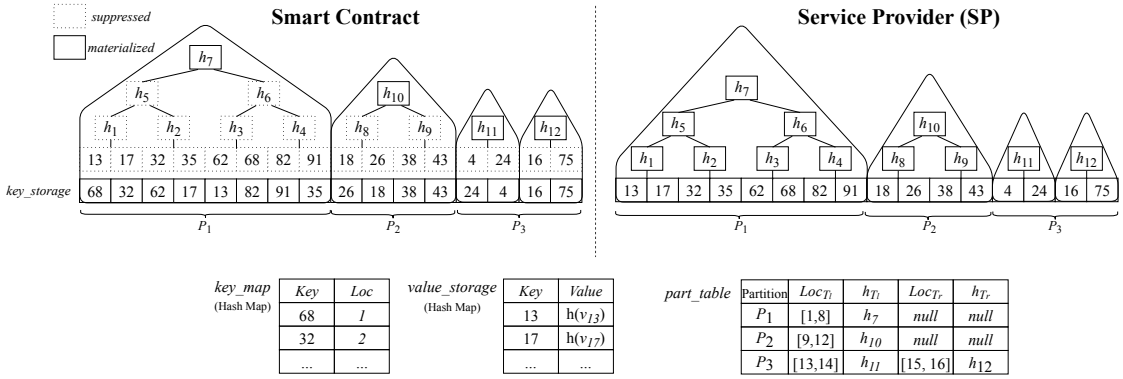


Fig. 4. Overall Structure of GEM^2-tree with Hybrid Storage

inserted objects. The benefit is twofold. On the one hand, a new object can always be inserted into the smaller SMB-trees, which is more gas-efficient. On the other hand, the objects indexed by the SMB-trees can be merged into the MB-tree *in batch* to optimize the update cost. The MB-tree structure is the same as that introduced in Section IV-A except that it is maintained by batched updates via merge operations (to be detailed in Section V-B).

Regarding the structure-suppressed SMB-trees, recall that the internal structure of an SMB-tree needs to be re-built for each object insertion, in order to update the root hash. Thus, to reduce the update cost, we organize the storage space into a set of exponentially-sized partitions. For each partition, up to two SMB-trees are maintained and they can be gracefully merged with more insertions. Note that the partitions are *logical* in the sense that they will dynamically change along with merges. This design has several advantages. First, as new object insertions can be directed to the smallest partition, less data need to be read and computed during the root hash update. Second, there is no need to physically re-arrange the objects after they are written into the storage, which is critical to save the gas cost. Third, this also significantly saves the maintenance cost on the SP side as it does not need to re-build the tree structure over the *entire* dataset for *each* object insertion. Forth, this ensures that the total number of partitions is $O(\log N)$, which will benefit the query processing.

Fig. 4 shows an example of the GEM^2-tree, where the fully-structured MB-tree is omitted for clarity. Besides *key_storage* and *value_storage*,[3] it consists of the following components: (i) a set of SMB-trees, one or two for each partition; (ii) an auxiliary partition index table (denoted as *part_table*); and (iii) a mapping between search key and storage location (denoted as *key_map*). The first two components are shared between the blockchain and the SP, while the last one is present only in the blockchain. It is worth noting that the search keys in the blockchain remain unsorted to reduce the gas cost; they are essentially stored in the order of insertions. Also, while the SMB-tree structures are suppressed in the blockchain, they are fully *materialized* on the SP side to support efficient query processing. The purpose of the *part_table* is to track how the storage space is partitioned. For each partition, we can maintain up to two SMB-trees (denoted as T_l and T_r, respectively). The *part_table* keeps the following information for each SMB-tree in each partition: (i) the storage location range (Loc) and (ii) the root hash (h). For example, in Fig. 4, T_l in P_1 corresponds to the objects stored in locations [1-8], T_r in P_1 is empty; in P_3, T_l and T_r correspond to the objects stored in locations [13-14] and [15-16], respectively. Here, the root hash is slightly different from the normal MB-tree as we also encode the key boundaries into it. For example, in Fig. 4, the root hash for tree T_l in P_1 is $h_7 = \mathsf{h}(13||91||\,\mathsf{h}(h_5||h_6))$. The extra boundary information can help the SP to prune the entire tree during the query processing, and thus improve the query performance. Finally, the *key_map* maintains an index of the storage location for each search key. It will be used during the update, to be explained in the next section.

[3]As explained in Section III, only the search keys and hashes of the objects are stored in the blockchain. We do not distinguish objects and object hashes, when the context is clear.

B. GEM^2-tree Maintenance

There are three maintenance operations for the GEM^2-tree: (i) insertion; (ii) updating; (iii) deletion. The deletion operation can be seen as updating the data object with a *dummy* one. Therefore, we focus on the insertion and updating operations only. For ease of illustration, we denote the partition for the fully-structured MB-tree as P_0 and the rest of partitions as $P_1, P_2, \cdots, P_{max}$. Let M be the maximum size of the smallest SMB-tree, i.e., the one in P_{max}. The size of each partition is thus $b_1 \cdot 2^{max-1} \cdot M, \cdots, b_{max-2} \cdot 4M, b_{max-1} \cdot 2M, b_{max} \cdot M$, where b_i is 1 or 2 depending on the number of SMB-trees existing in P_i.

Insertion. Algorithm 1 describes the insertion procedure. Whenever a new object arrives, it will be directed to the partition P_{max}. If the partition is not full (i.e., its size is less than $2M$), the object will be simply inserted into the current SMB-tree (lines 1–11). Otherwise, if the partition is full, a new SMB-tree is created with the object and a merge process is invoked to merge the two existing SMB-trees into a bigger SMB-tree of size $2M$, which will then be assigned to the preceding partition P_{max-1} (lines 13–17). If $max - 1$ is less than one, it means the corresponding partition does not exist yet. Thus, we need to increment max and create a new partition (line 14). The merge process is

Algorithm 1 GEM^2-Tree Insert(key, $value$)

Input Search key *key*, Data value *value*

1: $loc \leftarrow key\_storage.length + 1$;
2: $key\_map[key] \leftarrow loc$;
3: $key\_storage[loc] \leftarrow key$;
4: $value\_storage[key] \leftarrow \mathsf{h}(value)$;
5: **if** $P_{max} = null$ **then**
6: $P_{max}.Loc_{T_l} \leftarrow [1, M]$;
7: $P_{max}.Loc_{T_r} \leftarrow [M+1, 2M]$;
8: **if** $loc \in P_{max}.Loc_{T_l}$ **then**
9: $P_{max}.T_l \leftarrow$ BuildSMBTree($P_{max}.Loc_{T_l}$);
10: **else if** $loc \in P_{max}.Loc_{T_r}$ **then**
11: $P_{max}.T_r \leftarrow$ BuildSMBTree($P_{max}.Loc_{T_r}$);
12: **else**
13: $ret \leftarrow$ Merge(P_{max});
14: **if** $ret = true$ **then** $max \leftarrow max + 1$;
15: $P_{max}.Loc_{T_l} \leftarrow [loc, loc + M - 1]$;
16: $P_{max}.Loc_{T_r} \leftarrow [loc + M, loc + 2M - 1]$;
17: $P_{max}.T_l \leftarrow$ BuildSMBTree($P_{max}.Loc_{T_l}$);

Algorithm 2 GEM^2-Tree Merge(P_i)

Input Partition P_i
Output Whether to increment max flag ret

1: **if** $i = 1$ **then**
2: **if** $P_1.length < S_{max}$ **then**
3: $P_1.Loc_{T_l} \leftarrow P_1.Loc_{T_l} \cup P_1.Loc_{T_r}$;
4: $P_1.T_l \leftarrow$ BuildSMBTree($P_1.Loc_{T_l}$);
5: Empty $P_1.T_r$; $ret \leftarrow true$;
6: **else**
7: Bulk insert the data in P_1 to P_0;
8: Empty P_1; $ret \leftarrow false$;
9: **else if** $P_{i-1}.T_r = null$ **then**
10: $P_{i-1}.Loc_{T_r} \leftarrow P_i.Loc_{T_l} \cup P_i.Loc_{T_r}$;
11: $P_{i-1}.T_r \leftarrow$ BuildSMBTree($P_{i-1}.Loc_{T_r}$);
12: Empty P_i; $ret \leftarrow false$;
13: **else**
14: $ret \leftarrow$ Merge($i - 1$);
15: **if** $ret = true$ **then**
16: $P_i.Loc_{T_l} \leftarrow P_i.Loc_{T_l} \cup P_i.Loc_{T_r}$;
17: $P_i.T_l \leftarrow$ BuildSMBTree($P_i.Loc_{T_l}$);
18: Empty $P_i.T_r$; $ret \leftarrow true$;
19: **else**
20: $P_{i-1}.Loc_{T_l} \leftarrow P_i.Loc_{T_l} \cup P_i.Loc_{T_r}$;
21: $P_{i-1}.T_l \leftarrow$ BuildSMBTree($P_{i-1}.Loc_{T_l}$);
22: Empty P_i; $ret \leftarrow false$;
23: **return** ret;

detailed in Algorithm 2. It may take place recursively if the current partition is full and needs to make room for the newly merged SMB-tree. To avoid maintaining too many objects in a single SMB-tree that incurs high maintenance cost as discussed in Section IV-B, we set an upper bound, S_{max}, on the SMB-tree size. If the size of each SMB-tree to be merged exceeds $S_{max}/2$, instead of merging them, they will be bulk inserted into the fully-structured MB-tree P_0. This insertion procedure is the same for the smart contract and the SP except two differences: (i) instead of $value$, $\mathsf{h}(value)$ is stored in the blockchain; (ii) the construction of the SMB-trees in the smart contract, with internal nodes suppressed and key values unsorted, is carried out on the fly.

Updating. In contrast to the insertion operation, the updating operation replaces the value of an existing key with a new value. In this scenario, the GEM^2-tree structure remains unchanged. We only need to locate the corresponding partition for the updated object and recompute the root hash of the corresponding MB-tree or SMB-tree. The procedure is described in Algorithm 3. Recall that a nice property of the GEM^2-tree is that the storage location of each search key is fixed once it is stored in the blockchain, while the (logical) partitions will dynamically change with subsequent insertions and merges. Thus, we first find the storage location of the search key by checking the key_map (line 2). Then, we invoke the function, *LocatePartition*, with the storage location to identify the partition that contains the search key (line 3). After that, the corresponding tree is updated (lines 4–10).

Algorithm 3 GEM^2-Tree Update(key, $value$)

Input Search key key, Update value $value$

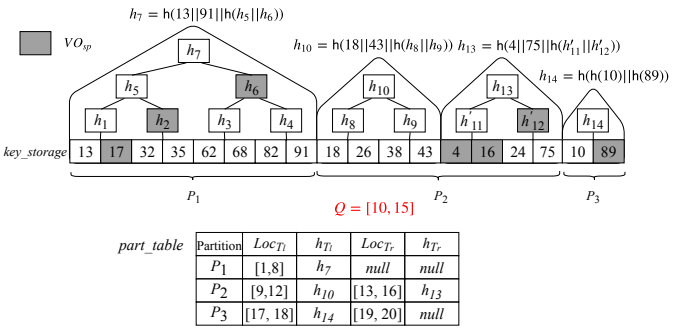
1: $value\_storage[key] \leftarrow \mathsf{h}(value)$;
2: $loc \leftarrow key\_map[key]$;
3: $p \leftarrow$ LocatePartition(loc, max);
4: **if** $p = 0$ **then**
5: Update MB-tree P_0 using $\langle key, value \rangle$;
6: **else**
7: **if** $loc \in P_p.Loc_{T_l}$ **then**
8: $P_p.T_l \leftarrow$ BuildSMBTree($P_p.Loc_{T_l}$);
9: **else**
10: $P_p.T_r \leftarrow$ BuildSMBTree($P_p.Loc_{T_r}$);

Algorithm 4 LocatePartition(loc, max)

Input Storage location loc, # partitions max
Output Partition index p

1: $p \leftarrow max$;
2: $[max\_start, max\_end] \leftarrow P_{max}.Loc_{T_r}$;
3: $len \leftarrow max\_end$; $cap \leftarrow 2M$;
4: **while** $p > 0$ **do**
5: **if** $len \mod cap = 0$ **then** ▷ There are two SMB-trees
6: **if** $loc \in [len - cap + 1, len]$ **then return** p;
7: $len \leftarrow len - cap$;
8: **else** ▷ There is only one SMB-tree
9: **if** $loc \in [len - cap/2 + 1, len]$ **then return** p;
10: $len \leftarrow len - cap/2$;
11: $p \leftarrow p - 1$; $cap \leftarrow 2 \cdot cap$;
12: **return** 0;

To implement the function *LocatePartition*, the simplest way is to check the $part\_table$ since it records the location range of each partition. However, this method is gas-inefficient as the whole table may need to be accessed in the worst case. To reduce the gas cost, we propose a more efficient algorithm that only needs to access the partition P_{max}. As detailed in Algorithm 4, after retrieving the location range of P_{max}, we search the partition from P_{max} to P_1 with respect to the maximum capacity of each partition. Since not all partitions contain two SMB-trees, we employ a mod operation to check whether or not the current partition contains two SMB-trees. If so, the mod result must be zero. For example, in Fig. 4, suppose we want to identify the partition for location 9. The initial space length is 16 and the maximum capacity of P_3 is 4, by checking 16 mod 4 = 0, we know that P_3 has two SMB-trees and hence spans from location 13 to 16. So location 9 is not in P_3. Next, the space length is reduced to 12 and



part_table

Partition	Loc_{Tl}	h_{Tl}	Loc_{Tr}	h_{Tr}
P_1	[1,8]	h_7	*null*	*null*
P_2	[9,12]	h_{10}	[13, 16]	h_{13}
P_3	[17, 18]	h_{14}	[19, 20]	*null*

Fig. 5. GEM^2-tree in the SP after Insertion

Algorithm 5 Authenticated Query with GEM^2-tree (by SP)

Input Query range Q, GEM^2-tree $\mathcal{T}$
Output Query result R, Verification object VO_{sp}
1: $(r_0, vo_0) \leftarrow$MBTreeRangeQuery$(Q, P_0.T)$;
2: Append r_0 to R and vo_0 to VO_{sp};
3: **for** each P_i in $\mathcal{T}.part\_table$ **do**
4: $(r_i.l, vo_i.l) \leftarrow$MBTreeRangeQuery$(Q, P_i.T_l)$;
5: $(r_i.r, vo_i.r) \leftarrow$MBTreeRangeQuery$(Q, P_i.T_r)$;
6: Append $\langle r_i.l, r_i.r\rangle$ to R, $\langle vo_i.l, vo_i.r\rangle$ to VO_{sp};
7: **return** $\langle R, VO_{sp}\rangle$;

we proceed to check P_2, whose maximum capacity is 8. By checking 12 mod 8 $\neq$ 0, P_2 has only a single SMB-tree and thus spans from location 9 to 12. Hence, we can know that location 9 is in P_2. If the location is not found in any SMB-tree partition, then we can conclude that it resides in the fully-structured MB-tree P_0.

Example. We use Fig. 4 and Fig. 5 as an example to illustrate the maintenance of GEM^2-tree. Suppose we want to insert two new keys 10 and 89 into the GEM^2-tree shown in Fig. 4. First, for the key 10, we find that the smallest partition P_3 is full. Therefore, we create a new SMB-tree containing the key 10. Meanwhile, we merge the two existing SMB-trees of P_3 into the preceding partition P_2. Since P_2's right SMB-tree T_r is empty, the merged tree will be put there and P_2's location range is extended. Next, for the key 89, it will be simply inserted into the SMB-tree in the new P_3 since is not full. As for data updates, suppose if the value of the key 26 is updated, we check the key_map and invoke the function *LocatePartition* to locate its partition P_2. After that, the value is updated and the corresponding root hash is updated by reconstructing the SMB-tree with the updated value.

C. Authenticated Query Processing

In this section, we discuss how to process authenticated queries over the hybrid-storage blockchain with our proposed GEM^2-tree. In the range query scenario, the client submits a query range $Q = [lb, ub]$. In turn, the SP returns all the objects lying in the range $[lb, ub]$, together with the proof VO_{sp}. Since the GEM^2-tree consists of one normal MB-tree and multiple SMB-trees, with each of them perhaps contributing to the query result, the SP is required to traverse all these trees and process the range query on them individually. After that, the SP combines the result objects and VO for each of these trees to generate the final query result and VO_{sp}. The

Algorithm 6 Result Verification with GEM^2-tree (by Client)

Input Query range Q, Query result R, VO_{sp} from the SP, VO_{chain} form the blockchain
Output Whether the verification is passed
1: Verify VO_{chain} w.r.t. the blockchain;
2: **for** $\langle r_i, vo_i\rangle$ in Q, R **do**
3: $T_i \leftarrow$ MB-tree root from VO_{chain} w.r.t. $\langle r_i, vo_i\rangle$;
4: $stat \leftarrow$ MBTreeVerify(r_i, vo_i, T_i);
5: **if** $stat = false$ **then return** $false$;
6: **return** $true$;

overall query processing procedure on the SP side is presented in Algorithm 5. First, *MBTreeRangeQuery* is invoked for the fully-structured MB-tree corresponding to the P_0 partition (lines 1–2). Then, it is invoked for both the left and right SMB-trees of each remaining partition (lines 3–6).

The *MBTreeRangeQuery* procedure is similar to that of the normal MB-tree range query. Here, we give a detailed description of the process. First, the SP checks whether or not the query range overlaps with the boundaries of the current tree root. If there is no overlap, it means that the current tree does not contribute to the query result. In this case, the tree root hash, which encodes the boundary information, can be used directly as the VO and the procedure is terminated. Otherwise if they overlap, the range query can be executed as a breadth-first search. Starting from the root node, if a non-leaf node intersects the query range, it will be branched with its subtree further explored; if a non-leaf node has no intersection with the query range, its hash will be added as part of the VO. When a leaf node is reached, the SP will check each underlying object. The objects which fall inside the query range will be added to the query result, while the hashes of the other objects will be appended to the VO. Note that the boundary search keys r_{lb}^- and r_{ub}^+, which are immediately outside the query range, should also be included in the VO to prove the completeness.

On the client side, the verification process is composed of two steps, namely retrieving VO_{chain} and result verification. During the VO_{chain} retrieval, the client retrieves from the blockchain the Merkle roots of all the trees in the GEM^2-tree. VO_{chain} can be verified by the client using the blockchain consensus protocol with respect to the latest block. With the verified VO_{chain}, the client can then execute *MBTreeVerify* for each tree in the GEM^2-tree to establish the soundness and completeness of the query result. The procedure is similar to that of the MB-tree. The client checks the VO_{sp} for each tree in two aspects:

- **Soundness Check.** The client reconstructs the tree's root hash using the query result R and the hashes of the sibling leaf nodes and adjacent non-leaf nodes in VO_{sp}. The check is passed if the reconstructed root hash is identical to the corresponding root hash obtained from VP_{chain}.
- **Completeness Check.** There are two cases. If the current tree range does not intersect with the query range, the client can ensure that there is no missing result by checking the boundary information with respect to the query range. Otherwise, the client can establish the completeness by checking the boundary search keys r_{lb}^- and r_{ub}^+.

The algorithm for result verification is summarized in Algorithm 6.

Example. Fig. 5 gives an example of authenticated query processing with the GEM^2-tree. Consider a range query $Q = [10, 15]$. The SP traverses all the MB-tree and SMB-trees. For partition P_1, there is only one SMB-tree and its key boundaries $[13, 91]$ overlap the query range. The result contains the object with key 13 and $vo_1.l$ consists of $\{17, [13, 91], h_2, h_6\}$. For partition P_2, the key boundaries of the left tree T_l (i.e., $[18, 43]$) do not overlap the query range. Therefore, the SP computes $\mathsf{h}(h_8||h_9)$, and $vo_2.l$ consists of $\{[18, 43], \mathsf{h}(h_8||h_9)\}$. The right tree T_r in P_2 is traversed as the key boundaries $[4, 75]$ overlap the query range, which generates $vo_2.r = \{4, 16, [4, 75], h_{12}\}$. Finally for partition P_3, the object with key 10 will be returned as the result and $\{89\}$ is constructed as $vo_3.l$. Combining everything together, the query result $R = \{10, 13\}$ and $VO_{sp} = \{vo_1.l, vo_2.l, vo_2, r, vo_3.l\}$ are sent to the client. During the result verification, the client first obtains the verified $VO_{chain} = \{h_7, h_{10}, h_{13}, h_{14}\}$ from the blockchain. Next, each tree root is reconstructed as the following: $h_7^* = \mathsf{h}(13||91||\,\mathsf{h}(\mathsf{h}(\mathsf{h}(13||17)||h_2)||h_6))$, $h_{10}^* = \mathsf{h}(18||43||\,\mathsf{h}(h_8||h_9))$, $h_{13}^* = \mathsf{h}(4||75||\,\mathsf{h}(\mathsf{h}(4||16)||h'_{12}))$, and $h_{14}^* = \mathsf{h}(10||89||\,\mathsf{h}(10||89))$. With each of them verified against VO_{chain} and boundary search keys checked against the query range, both the soundness and completeness of the query result can be established.

D. Comparing with Log-Structured Merge-tree

The *Log-Structured Merge-tree* (LSM-tree) is a data structure proposed to optimize the I/O cost in the write-dominant environments [12]. Its modern variations [13] usually implement a multilevel structure, which also partitions the data space in an exponential fashion. In this section, we highlight the differences between our proposed GEM^2-tree and the LSM-tree and discuss why the LSM-tree would fail in our problem.

- *LSM-tree requires to maintain long sorted lists.* The LSM-tree requires the lists sorted at all levels, using a merge-sort like algorithm. During its merge process, a newly sorted list is created while the old lists are discarded. This would be highly inefficient in the case of the smart contract as too many writes will be incurred. In comparison, our GEM^2-tree avoids maintaining sorted lists. The data remains unsorted in the blockchain storage, while the tree structures are computed on the fly.
- *LSM-tree nodes are materialized.* As analyzed in Section IV, materializing the tree nodes would incur high overhead during updates.
- *There is no upper bound of the number of levels in the LSM-tree*. With the size of the level enlarged exponentially, the cost of merging two trees is increased proportionally. This is undesirable since a merge operation in the LSM-tree requires building a new fully sorted list and its corresponding tree structure, which yields a complexity of $O(N)$. In contrast, our GEM^2-tree will fall back to a normal MB-tree with batched updates in $O(\log N)$ complexity when the size of the largest partition exceeds a certain threshold.
- *The update operations of the LSM-tree and the GEM^2-tree are different*. The update operation of the LSM-tree is done by appending a new record with a duplicate key. The outdated records are discarded only when the compaction process is invoked. In contrast, the GEM^2-tree employs in-place update by locating the partition of the index and updating the corresponding record directly, which is more efficient.

E. Security Analysis

In this section, we perform a security analysis on our proposed GEM^2-tree and its associated query authentication algorithm. We start by presenting a formal definition of our security notion.

Definition 1 (Secure)**.** *The query authentication algorithm is sound and complete if for all PPT adversaries, the probability is negligible in the following experiment:*

- *an adversary $\mathcal{A}$ selects a dataset $\mathbb{D}$;*
- *the authentication algorithm constructs the ADS and its corresponding VO_{chain} based on $\mathbb{D}$ and sends them to $\mathcal{A}$;*
- *$\mathcal{A}$ outputs a tuple of range query Q, result R, and VO_{sp}.*

The adversary $\mathcal{A}$ succeeds if VO_{sp} passes the verification with respect to VO_{chain} and satisfies the condition: $\{r_i | r_i \notin Q(\mathbb{D}) \wedge r_i \in R\} \neq \varnothing \vee \{r_j | r_j \in Q(\mathbb{D}) \wedge r_j \notin R\} \neq \varnothing$.

The above definition states that a malicious SP could convince the user of an incorrect or incomplete answer with at most a negligible probability. We now show that our proposed query authentication algorithm indeed satisfies the desired security requirement.

Theorem 1. *Our proposed authenticated query algorithm based on the GEM^2-tree is secure if the underlying hash function is collision resistant.*

Proof. We prove this theorem by contradiction.

Case 1: $\{r_i | r_i \notin Q(\mathbb{D}) \wedge r_i \in R\} \neq \varnothing$. This means that there is an object in R which is not originated from $\mathbb{D}$. Since the client will reconstruct the hash root of the MB-tree/SMB-tree in which r_i lies and compare it against the hash root in VO_{chain}, such a tampered result means that there exist two MB-trees/SMB-trees with different objects but the same hash root. This implies a successful collision of the underlying hash function, which leads to a contradiction to our assumption.

Case 2: $\{r_j | r_j \in Q(\mathbb{D}) \wedge r_j \notin R\} \neq \varnothing$. This means that there is a valid answer missing from R. Since the client will verify the completeness with the boundary information of the entire tree or the boundary search keys which are adjacent to the query range for each subtree of the GEM^2-tree. A missing answer will inevitably lead to a hash collision for some MB-tree/SMB-tree. Then we arrive at a contradiction to the assumption. □

F. Cost Analysis

In this section, we perform a cost analysis for both the GEM^2-tree maintenance and authenticated query processing.

We assume that the database size N is larger than $2S_{max}$. This means that the fully-structured MB-tree always exists in P_0. It is also trivial to see that $S_{max} = 2^{max}M$ in this case.

ADS Maintenance Cost. First, we analyze the GEM^2-tree insertion cost. Let $P_{\text{Merge}}(i)$ be the probability of invoking the merge operation over the partition P_i. We have $P_{\text{Merge}}(max) = 1/(2M)$ and $P_{\text{Merge}}(i) = P_{\text{Merge}}(i+1)/2$. Further, we can derive that $P_{\text{Merge}}(1)$, the probability of the largest SMB-tree partition P_1 being inserted to the fully-structured MB-tree, is $1/(2^{max}M)$. Applying the cost analysis of the MB-tree/SMB-tree in Section IV, we can obtain the average cost of the GEM^2-tree insertion operation is:

$$\begin{aligned}
C^{\text{insert}}_{\text{GEM}^2\text{-tree}} =& C^{\text{insert}}_{\text{SMB-tree}}(M) + P_{\text{Merge}}(1)\big(C^{\text{insert}}_{\text{MB-tree}}(N-2S_{max})S_{max} \\
& - C_{\text{bshare}}(S_{max})\big) + \sum_{i=2}^{max} P_{\text{Merge}}(i)\cdot C^{\text{insert}}_{\text{SMB-tree}}(2^{max-i+1}M) \\
\approx& C_1 \log_F (N - 2^{max+1}M) + C_2 \cdot max^2 \\
& + C_3 \cdot max + C_4 \\
\text{where}\quad & C_1 = 2C_{sstore} + 2C_{supdate} + (2F+1)C_{ssload} + C_{hash} \\
& C_2 = \log 2 \cdot C_{mem}/2 \qquad C_3 = C_{ssload} + C_{hash}/F \\
& C_4 = 2C_{sstore} + MC_{ssload} + (1 - 2\log_F 2^{max}M)C_{update}
\end{aligned}$$

Here, the C_{bshare} is the cost saved by the bulk insertion of the largest SMB-trees, which can be approximated by $\log_F S_{max}$. It can be observed that the insertion complexity $C^{\text{insert}}_{\text{GEM}^2\text{-tree}}$ is $O(\log N)$ with respect to the database size. Compared with the normal MB-tree, our GEM^2-tree is able to trade some portion of the overhead of the MB-tree maintenance with that of the SMB-tree, which leads to a better performance.

Next, we analyze the cost of the update operation. Let $P_{\text{Update}}(i)$ be the probability of updating an object lying in partition P_i. Assuming that data updates take place uniformly throughout the whole space, we can get $P_{\text{Update}}(i) = 2^iM/N$ for $i \in [1, max]$ and $P_{\text{Update}}(0) = (N - 2S_{max})/N$. Moreover, it is easy to see that the update cost of the MB-tree and SMB-tree is as follows:

$$\begin{aligned}
C^{\text{update}}_{\text{MB-tree}} =& \log_F N(C_{supdate} + (F+1)C_{sload} + C_{hash}) + C_{supdate} \\
C^{\text{update}}_{\text{SMB-tree}} =& N\Big(C_{sload} + \log N \cdot C_{mem} + \frac{1}{F}C_{hash}\Big) + C_{supdate}
\end{aligned}$$

Thus, the average cost of the GEM^2-tree update operation is:

$$\begin{aligned}
C^{\text{update}}_{\text{GEM}^2\text{-tree}} =& P_{\text{Update}}(0)\cdot C^{\text{update}}_{\text{MB-tree}}(N - 2S_{max}) \\
& + \sum_{i=1}^{max} P_{\text{Update}}(i)\cdot C^{\text{update}}_{\text{SMB-tree}}(2^iM) \\
\approx& \frac{1}{N}\Big(C_5 \log_F (N - 2^{max+1}M)(N - 2^{max+1}M) \\
& + C_{supdate}(N - 2^{max+1}M) + C_6 2^{2max+2} max \\
& + C_7 2^{2max+2} + C_8 2^{max+1}\Big) \\
\text{where}\quad & C_5 = C_{supdate} + (F+1)C_{ssload} + C_{hash} \\
& C_6 = \log 2 \cdot M^2 C_{mem}/3 \\
& C_7 = M^2(C_{ssload} + C_{hash}/F)/3 \qquad C_8 = MC_{supdate}
\end{aligned}$$

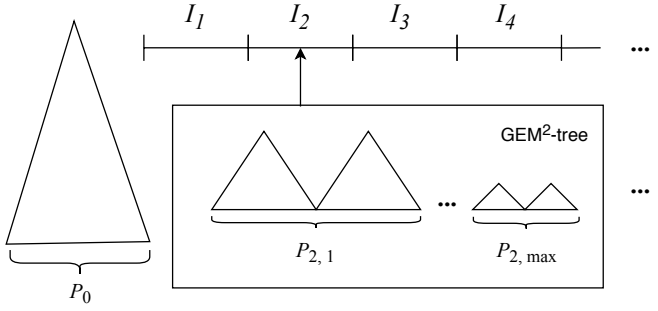


Fig. 6. Overall Structure of the GEM^{2*}-tree

Similar to the insertion, the update cost is in the complexity of $O(\log N)$.

Query Processing Cost. The cost of processing a query over a single MB-tree of size N is $C_{query} \cdot \log_F N$, where C_{query} is a constant denoting the query cost of a single node. Since the SP has to traverse all of subtrees inside the GEM^2-tree, whose sizes are $N - 2S_{max}$ for P_0 and $2^{max-i}M$ for $P_i, i \in [1, max]$. In the worst case, the SP computation cost and the size of VO_{sp} both are:

$$\begin{aligned}
C^{\text{query}}_{\text{GEM}^2\text{-tree}} =& C_{query}\cdot \log_F (N - 2S_{max}) + \sum_{i=1}^{max} C_{query}\cdot \log_F (2^{max-i}M) \\
=& C_{query}\Big(\log_F (N - 2^{max+1}M) \\
& + \frac{\log_F 2}{2}\cdot max^2 + (\log_F M - \frac{\log_F 2}{2})\cdot max\Big)
\end{aligned}$$

This is again in the complexity of $O(\log N)$. As for VO_{chain}, its size is linear to the number of partitions (i.e., max).

VI. Optimized GEM^2-tree

In this section, we present an optimized index called GEM^{2*}-tree, which can further reduce the gas consumption cost without sacrificing much in terms of the query overhead.

A. GEM^{2*}-tree Structure and Maintenance

The basic structure of the GEM^{2*}-tree is a two-level index as shown in Fig. 6. In the upper level, we split the search key domain into several regions $I_1, I_2, I_3, \cdots$. In order to achieve the maximum performance, the split is based on the underlying data distribution so that the keys expected to fall in each region I_i are the same. In the lower level, a GEM^2-tree is built for each I_i. It is worth noting that there is a slight difference between the GEM^2-tree constructed here and the standalone one. Instead of maintaining a fully-structured MB-tree P_0 for each GEM^2-tree corresponding to each I_i, there is only one single fully-structured MB-tree for the entire GEM^{2*}-tree. With the above design, the following benefits are expected:

- *More Gas Savings.* Based on the cost analysis in Section V-F, the reduction of the gas consumption of the GEM^2-tree compared with the normal MB-tree comes from the use of the SMB-trees in the small to medium-sized partitions. As the GEM^{2*}-tree maintains more SMB-trees while avoiding SMB-trees of too large size, thanks

Algorithm 7 Authenticated Query with GEM^{2*}-tree (by SP)

Input Query range $Q = [lb, ub]$, GEM^{2*}-tree $\mathcal{T}^*$
Output Query result R, Verification object VO_{sp}
1: $li \leftarrow \mathcal{T}^*.upper\_level$.BinarySearch($lb$);
2: $ui \leftarrow \mathcal{T}^*.upper\_level$.BinarySearch($ub$);
3: **for** i **in** $[li, ui]$ **do**
4: $\quad\langle r_i, vo_i\rangle \leftarrow$ GEM^2-tree Query$(Q, \mathcal{T}^*.lower_level[i])$;
5: $\quad$Append r_i to R, vo_i to VO_{sp};
6: $\langle r_0, vo_0\rangle \leftarrow$ MBTreeRangeQuery$(Q, \mathcal{T}^*.P_0)$;
7: Append r_0 to R, vo_0 to VO_{sp};

Algorithm 8 Result Verification with GEM^{2*}-tree (by Client)

Input Query range $Q = [lb, ub]$, Query result R, VO_{sp} from the SP, VO_{chain} from the blockchain
Output Whether the verification is passed
1: Verify VO_{chain} w.r.t. the blockchain;
2: $upper\_level \leftarrow$ GEM^{2*}-tree upper level from VO_{chain};
3: $li \leftarrow upper\_level$.BinarySearch($lb$);
4: $ui \leftarrow upper\_level$.BinarySearch($ub$);
5: **for** i **in** $[li, ui]$ **do**
6: $\quad vo_{chain,i} \leftarrow$ GEM^2-tree root in VO_{chain} for i-th region;
7: $\quad$Extract $\langle r_i, vo_i\rangle$ from $\langle R, VO_{sp}\rangle$ w.r.t. i-th region;
8: $\quad stat \leftarrow$ GEM^{2*}-tree Verify$(Q, r_i, vo_i, vo_{chain,i})$;
9: $\quad$**if** $stat = false$ **then return** $false$;
10: $T_0 \leftarrow$ MB-tree root from VO_{chain} w.r.t. P_0;
11: Extract $\langle r_0, vo_0\rangle$ from $\langle R, VO_{sp}\rangle$ w.r.t. P_0;
12: $stat \leftarrow$ MBTreeVerify(r_0, vo_0, T_0);
13: **if** $stat = false$ **then return** $false$;
14: **return** $true$;

to the split search key domain, it can contribute to more gas savings.

- *Efficient Query Processing Retained.* Although the GEM^{2*}-tree introduces more subtrees, the query performance is not much sacrificed. The reason is twofold. First, due to the space splitting in the upper level, not all of lower-level index trees need to be visited during the query processing. Moreover, each region I_i contains only a portion of the entire dataset, which leads to smaller trees that can help expedite query processing.

The maintenance of the GEM^{2*}-tree is straightforward. During data insertions or updates, we first locate the upper-level region based on the boundary information. Then, the corresponding GEM^2-tree in the lower level is updated accordingly using the procedure identical to the one introduced in Section V-B.

B. Authenticated Query Processing

The query processing and result verification algorithms with the GEM^{2*}-tree are similar to those of the GEM^2-tree. Algorithm 7 shows the authenticated query processing procedure. First, a binary search is used to locate the leftmost and rightmost upper-level regions which overlap the query range (lines 1–2). Then, the SP invokes Algorithm 5 for each low-level GEM^2-tree under the corresponding region (lines 3–5). Finally, the fully-structured MB-tree is searched (lines 6–7). In a similar manner, the verification procedure is presented in Algorithm 8. It consists of a binary search of the upper-level regions (lines 2–4), verifying the result for each GEM*-tree (lines 5–9), and verifying the result for the fully-structured MB-tree (lines 10–13).

VII. Performance Evaluation

In this section, we evaluate the performance of our proposed ADSs, namely GEM^2-tree and GEM^{2*}-tree.

A. Experimental Settings

We use the *Yahoo Cloud System Benchmark* (YCSB) [14] to generate synthetic datasets for performance evaluation. All the generated datasets contain 100M update records, in which each search key has a size of 4 bytes and each value has a size of 100 bytes. Two search key distributions, i.e., uniform distribution and zipfian distribution, are evaluated. In the latter, the zipfian constant is set to 0.8 to generate skewed datasets.

For our GEM^2-tree and GEM^{2*}-tree, the following settings are adopted. The maximum size of the smallest SMB-tree, M, is set to 8 as the word size in Ethereum is 32 bytes and the search key has a size of 4 bytes. The fanout of the MB-tree is set to 4, which is the maximum of f satisfying $(f-1) \times l_d + f \times l_p + l_p < 32$ bytes, where l_d and l_p are the sizes of the delimiters and pointers. The upper bound of the largest SMB-tree partition, S_{max}, is set to $2,048$, which is based on the cost analysis of the MB-tree and SMB-tree given in Section IV. Moreover, for the upper-level index of the GEM^{2*}-tree, the search key domain is split into 100 regions based on the key distribution.

In the experiments, a private Ethereum network using Geth[4] is deployed. The smart contract is implemented in Solidity. For each of the SP and the client, a desktop computer with Intel Core i7-7700K 4.2GHz CPU and 16GM RAM, running Ubuntu 18.04.1 LTS, is used. The query processing and result verification programs are written in Java. We choose SHA-3 as the cryptographic hash function in the implementation of all algorithms.

For comparison, two baseline algorithms, MB-tree [15] and LSM-tree [13], are also implemented. We measure the following metrics to evaluate the algorithms: (i) the blockchain's *gas* cost for ADS maintenance, (ii) the SP's query processing time, (iii) the size of the VO (including both V_{SP} and V_{chain}), and (iv) the client's result verification time.

B. Experimental Results

1) Gas Consumption for ADS Maintenance: Fig. 7 shows the average gas consumption with increasing database size. Clearly, our proposed GEM^2-tree and GEM^{2*}-tree are more efficient than the two baselines regardless of the data distribution. In particular, the LSM-tree is only able to support the database with up to $10,000$ objects. This is mainly because the merge cost in the LSM-tree grows exponentially with its level depth increasing, as discussed in Section V-D. As such, the LSM-tree is impractical to be maintained by the smart contract. Compared with the MB-tree, our solutions reduce the gas consumption by a factor of up to 4. The gas reduction comes from both the efficient SMB-trees and the bulk insertion

[4]https://github.com/ethereum/go-ethereum

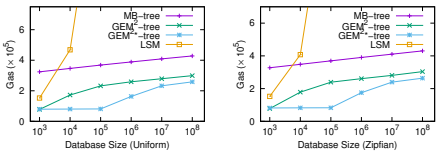


Fig. 7. Gas Consumption vs. Database Size

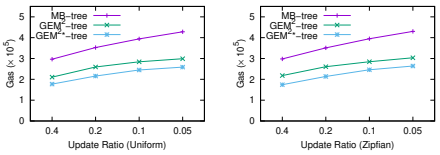


Fig. 8. Gas Consumption vs. Update Ratio

of objects into the MB-tree. Further, the GEM^{2*}-tree always consumes less gas than the GEM^2-tree. This is because the GEM^{2*}-tree contains more SMB-trees, which can help serve more objects in an efficient way. Moreover, thanks to the partitioning of the search key space, the objects bulk inserted from the SMB-trees into the MB-tree are more likely to be located in the same range, which makes the bulk insertion more efficient.

To further evaluate the performance with respect to data insertions vs. updates, we measure the average gas cost for the workloads with different update ratios. Starting with an existing database that contains $10,000,000$ objects, we send $90,000,000$ insertion or update requests to the smart contract. The update ratio is varied from 40% to 5%, which is equivalent to $36,000,000$ to $4,500,000$ update operations. The average gas cost is plotted in Fig. 8. Since the update cost is lower than the insertion cost, the less the update operations the more gas consumed. It can also be observed that in all cases tested, the GEM^2-tree achieves at least 30% gas reduction compared with the MB-tree. The performance of the GEM^{2*}-tree is even better, thanks to its higher capacity for maintaining the SMB-trees and the search key domain regions. Another interesting observation is that our solutions save more gas against the MB-tree when there are more insertion operations. This further demonstrates the advantages of our proposals.

2) Query Performance: We now investigate the query performance of the different algorithms. The results are shown in Fig. 9 and Fig. 10. In our experiments, the database size is fixed to be 100M and we vary the query selectivity from 1% to 10%. For each experiment, 50 range queries are randomly generated and the average performance results are reported. As can been seen, for all algorithms, all metrics increase monotonically with the query range regardless of the data distribution. Compared with the MB-tree, the GEM^2-tree retains the query performance in all cases tested, while the GEM^{2*}-tree is only slightly worse when the query range is large and/or the key distribution is skewed, due to the reasons discussed in Section VI-A. Combining with the previous experiments on the ADS maintenance, this demonstrates that our solutions are able to drastically reduce the maintenance cost with little penalty on the query processing performance.

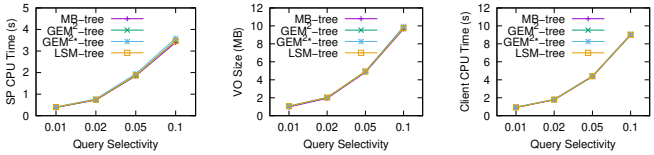


Fig. 9. Authenticated Query and Verification Performance (Uniform)

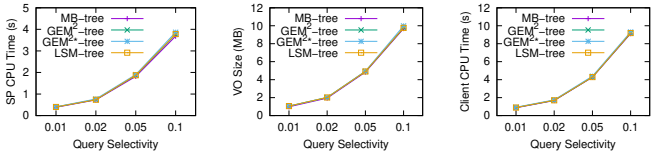


Fig. 10. Authenticated Query and Verification Performance (Zipfian)

VIII. Related Works

To the best of our knowledge, no studies exist that investigate gas-efficient data structures for authenticated queries in blockchain. In the following, we briefly survey related works and discuss relevant techniques.

A. Blockchain Technology

Blockchain has been a hot research area in recent years. Various issues have been studies, such as consensus algorithms [16], [17], storage designs [18], system security [19], [20], and privacy issues [21]–[24]. A benchmark framework for analyzing representative private blockchains was presented in [1]. Hu *et al.* [25] proposed a searchable encryption scheme over the blockchain with integrity assurance. But it is limited to file-level keyword search. Moreover, it does not investigate the indexing issue as only on-chain data are considered. More recently, Xu *et al.* [2] developed a novel vChain framework to enable verifiable queries over blockchain databases. To support dynamic data aggregation over arbitrary query attributes, an accumulator-based ADS scheme was proposed in [2]. In addition, some startups (e.g., [26], [27]) have proposed to expose a relational database frontend to the blockchain data storage. However, all these existing studies fail to consider the integrity issue when outsourcing query processing to off-chain storage services, which is the focus of this paper.

B. Authenticated Query Processing

There is a large body of research on authenticated query processing, verifying the integrity of query results produced by an untrusted service provider [15], [28]–[32]. There are two basic techniques for query authentication, namely digital signature chaining and Merkle Hash Tree (MHT). The former is a public-key message authentication scheme based on asymmetric cryptography. A digital signature is produced for each data object by the data owner using a private key. A client can verify the authenticity of a query result using the owner's public key and the object's signature. To establish the completeness of query results, chaining signatures are generated to capture the correlation of each object with its neighboring objects [28]. Signature chaining is simple, but it requires each object to be signed and thus cannot scale up to large datasets.

MHT [10], as discussed in Section II, solves the scalability issue using a hierarchical tree structure. MHT has been adapted to various index structures. Typical examples include the Merkle B-tree for relational data [15], the Merkle R-tree for spatial data [29], [33], [34], and the authenticated inverted index for text data [35]. It has also been extended to support authenticated join queries [36], distributed and shared data [30], [31]. Nevertheless, to the best of our knowledge, no previous works exist that study authenticated relational queries for data stored in a hybrid-storage blockchain.

IX. Conclusion

In this paper, we have studied the problem of authenticating range queries for databases stored in the hybrid-storage blockchain. The main challenge lies in how to design an ADS which can be efficiently maintained by the smart contract in the blockchain. By analyzing the performance of the existing solutions, we have proposed a novel gas-efficient ADS, called GEM^2-tree, that can significantly reduce the storage and computation costs of the smart contract. We have also developed an optimized ADS, called GEM^{2*}-tree. It further saves the maintenance cost by splitting the data domain and introducing a two-level structure. Analytical models and empirical results have substantiated the robustness and efficiency of our proposed solutions.

This paper opens up a new direction for blockchain research. Specifically, many previous query authentication techniques require new design under the gas performance model. For example, it will be interesting to explore how to design gas-efficient data structures for other authenticated queries, such as keyword and aggregation queries.

Acknowledgements

This work was supported by Research Grants Council of Hong Kong under GRF Projects 12201018, 12244916, and CRF Project C1008-16G. Yuzhe Tang was partially supported by the National Science Foundation under Grant CNS-1815814, CUSE grant 286056 and a gift from Intel.

References

[1] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan, "BLOCKBENCH: A framework for analyzing private blockchains," in *Proc. SIGMOD*, 2017.

[2] C. Xu, C. Zhang, and J. Xu, "vChain: Enabling verifiable boolean range queries over blockchain databases," in *Proc. SIGMOD*, 2019.

[3] L. S. Sankar, M. Sindhu, and M. Sethumadhavan, "Survey of consensus protocols on blockchain applications," in *Proc. ICACCS*, 2017, pp. 1–5.

[4] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[5] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[6] R. Matzutt, J. Hiller, M. Henze, J. H. Ziegeldorf, D. Müllmann, O. Hohlfeld, and K. Wehrle, "A quantitative analysis of the impact of arbitrary blockchain content on bitcoin," in *Proc. FC*, 2018.

[7] B. Liu, X. L. Yu, S. Chen, X. Xu, and L. Zhu, "Blockchain based data integrity service framework for iot data," in *Proc. ICWS*, 2017, pp. 468–475.

[8] G. Ayoade, V. Karande, L. Khan, and K. Hamlen, "Decentralized iot data management using blockchain and trusted execution environment," in *Proc. IRI*, 2018, pp. 15–22.

[9] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proc. SIGMOD*, 2006, pp. 121–132.

[10] R. C. Merkle, "A certified digital signature," in *Proc. CRYPTO*, 1990, pp. 218–238.

[11] K. Ren, C. Wang, and Q. Wang, "Security challenges for the public cloud," *IEEE Internet Computing*, vol. 16, no. 1, pp. 69–73, 2012.

[12] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, Jun 1996.

[13] Y. Li, B. He, Q. Luo, and K. Yi, "Tree indexing on flash disks," in *Proc. ICDE*, March 2009, pp. 1303–1306.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. ACM symposium on Cloud computing*, 2010, pp. 143–154.

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proc. SIGMOD*, 2006.

[16] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A scalable blockchain protocol," in *USENIX NSDI*, 2016, pp. 45–59.

[17] G. Pirlea and I. Sergey, "Mechanising blockchain consensus," in *ACM SIGPLAN Int'l Conf. Certified Programs and Proofs*, 2018, pp. 78–90.

[18] S. Wang, T. T. A. Dinh, Q. Lin, Z. Xie, M. Zhang, Q. Cai, G. Chen, W. Fu, B. C. Ooi, and P. Ruan, "ForkBase: An efficient storage engine for blockchain and forkable applications," *Proc. VLDB*, vol. 11, no. 10, pp. 1137 – 1150, 2018.

[19] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel, "Betrayal, distrust, and rationality: Smart counter-collusion contracts for verifiable cloud computing," in *Proc. CCS*, 2017, pp. 211–227.

[20] J. Camenisch, M. Drijvers, and M. Dubovitskaya, "Practical uc-secure delegatable credentials with attributes and their application to blockchain," in *Proc. CCS*, 2017, pp. 683–699.

[21] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *Proc. S&P*, 2014, pp. 459–474.

[22] E. Cecchetti, F. Zhang, Y. Ji, A. Kosba, A. Juels, and E. Shi, "Solidus: Confidential distributed ledger transactions via pvorm," in *Proc. CCS*, 2017, pp. 701–717.

[23] G. Ateniese, B. Magri, D. Venturi, and E. Andrade, "Redactable blockchain–or–rewriting history in bitcoin and friends," in *Proc. EuroS&P*, 2017, pp. 111–126.

[24] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *2016 IEEE symposium on security and privacy (SP)*. IEEE, 2016, pp. 839–858.

[25] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren, "Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization," in *Proc. INFOCOM*, 2018.

[26] B. M. Platz, A. Filipowski, and K. Doubleday, "Flureedb, a practical decentralized database," 2017. [Online]. Available: https://flur.ee/assets/pdf/flureedb_whitepaper_v1.pdf

[27] BigchainDB GmbH, "Bigchaindb 2.0 the blockchain database," 2018. [Online]. Available: https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf

[28] H. Pang and K.-L. Tan, "Authenticating query results in edge computing," in *Proc. ICDE*, 2004.

[29] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios, "Authenticated indexing for outsourced spatial databases," *The VLDB Journal*, no. 3, pp. 631–648, 2008.

[30] Q. Chen, H. Hu, and J. Xu, "Authenticated online data integration services," in *Proc. SIGMOD*, 2015.

[31] C. Xu, J. Xu, H. Hu, and M. H. Au, "When query authentication meets fine-grained access control: A zero-knowledge approach," in *Proc. SIGMOD*, 2018.

[32] C. Xu, Q. Chen, H. Hu, J. Xu, and X. Hei, "Authenticating aggregate queries over set-valued data with confidentiality," *IEEE Transactions on Knowledge and Data Engineering*, vol. 30, no. 4, pp. 630–644, 2018.

[33] M. L. Yiu, E. Lo, and D. Yung, "Authentication of moving kNN queries," in *Proc. ICDE*, 2011.

[34] C. Zhang, C. Xu, J. Xu, and B. Choi, "Distributed kNN query authentication," in *Proc. MDM*, 2018, pp. 167–176.

[35] H. Pang and K. Mouratidis, "Authenticating the query results of text search engines," in *Proc. VLDB*, 2008.

[36] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated join processing in outsourced databases," in *Proc. SIGMOD*, 2009.