Authenticated Aggregate Queries with Boolean Range Predicates on Blockchains

Weijie Sun
HKUST
Hong Kong SAR, China
wsunan@cse.ust.hk

Zihuan Xu*
Shenzhen Institute of Computing Sciences
Shenzhen, China
xuzihuan@sics.ac.cn

Wangze Ni†
Zhejiang University
Hangzhou, China
niwangze@zju.edu.cn

Lei Chen
HKUST(GZ) & HKUST
Guangzhou & Hong Kong SAR, China
leichen@cse.ust.hk

Peng Cheng
Tongji University
Shanghai, China
cspcheng@tongji.edu.cn

Chen Jason Zhang
The Hong Kong Polytechnic University
Hong Kong SAR, China
jason-c.zhang@polyu.edu.hk

ABSTRACT

Blockchains have gained wide adoption for secure data processing. As blockchain data volumes grow, the demand for efficient data analysis, especially aggregate queries, becomes increasingly critical. However, current blockchains lack native support for efficient analytical query processing, forcing users to either maintain full replicas or rely on third-party services without integrity guarantees.

In this paper, we propose an efficient framework, Merkle Bloom Filter Tree (MBFT), for authenticated aggregate queries that combine boolean keywords and range predicates on blockchains. At its core is a Bloom filter-based authenticated data structure that supports both types of predicates, constructed per block for efficient transaction indexing. For temporal predicates, we optimize time window queries through value pruning and block consolidation. We design a novel Merge Bloom Filter (MBF) for space-efficient handling of dynamic sets during query authentication. We provide a theoretical analysis of the storage overhead caused by the Bloom filter's false positive rates. Our framework employs data sketches to support various aggregate operations. Extensive experiments demonstrate that MBFT has improved the query speed by up to 286× compared to state-of-the-art authenticated query solutions.

PVLDB Reference Format:
Weijie Sun, Zihuan Xu, Wangze Ni, Lei Chen, Peng Cheng, and Chen Jason Zhang. Authenticated Aggregate Queries with Boolean Range Predicates on Blockchains. PVLDB, 18(10): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/SwJay/mbft.

*Zihuan Xu is the corresponding author.
†Wangze Ni is also with The State Key Laboratory of Blockchain and Data Security; Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security.

Proceedings of the VLDB Endowment, Vol. 18, No. 10 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

With the popularity of Bitcoin [26], the decentralized and immutable nature of blockchains over fault-tolerant protocols [4] has revolutionized various domains like finance [32], supply chains [20] and crowdsourcing [18], etc.. As these blockchain-based applications thrive, massive volumes of valuable data are being generated, increasing user demands for querying and analyzing data managed in blockchains. In particular, the aggregate query, widely applied in database systems [8, 16], can provide users with valuable insights by efficiently summarizing massive blockchain data into meaningful statistics. Consider the following example:

Example 1.1. Fig. 1 shows three example aggregate queries with keyword and range predicates in blockchain-based applications:
(1) In crowdsourcing, Q_1 highlights how a contributor with address "*0x0af8*" tracks their highest single-task reward within a week.
(2) In finance, Q_2 demonstrates how a client estimates the total transferred amount of major transactions (value >100) from their address "*0x1b3e*" within a month.
(3) In supply chains, Q_3 describes how a retailer with address "*0x2ce7*" counts purchase transactions (value between 1000 and 5000) with a supplier at address "*0x3db6*" within a year. □

Q_1:
```
SELECT MAX(value) FROM txns WHERE receiver='0x0af8' AND
timestamp>='2024-01-01' AND timestamp<='2024-01-07';
```
Q_2:
```
SELECT SUM(value) FROM txns WHERE sender='0x1b3e' AND value>100
AND timestamp>='2024-01-01' AND timestamp<='2024-01-31';
```
Q_3:
```
SELECT COUNT(*) FROM txns WHERE sender='0x2ce7' AND
receiver='0x3db6' AND value>1000 AND value<5000 AND
timestamp>='2023-01-01' AND timestamp<='2023-12-31';
```

Figure 1: Example Aggregate Queries on Blockchains.

Despite the importance of predicated aggregated query processing, native blockchains still lack efficient support for such queries. Consequently, users can either (1) run a full node with an external query engine to maintain complete blockchain data, which incurs significant storage (*e.g.*, over 600 GB for Bitcoin [39] and 1 TB for Ethereum [40]) and computation overhead; or (2) resort to centralized third-party blockchain databases (*e.g.*, BigQuery [6], BigchainDB [15]) whose query execution must be trusted. It contradicts blockchain's Byzantine setting, as results cannot be independently verified to guarantee integrity [34, 38].

Authenticated query, as a practical solution, balances space efficiency and data integrity. Fig. 2 illustrates the *authenticated query processing* in blockchains, where full nodes store complete blocks, and light nodes maintain only block headers. Each block stores a *Merkle Hash Tree* (MHT) as its *authenticated data structure* (ADS), with the root hash embedded in the block header. The full node executes queries and returns the result with a *verification object* (VO) from the ADS for the light node to verify result integrity.

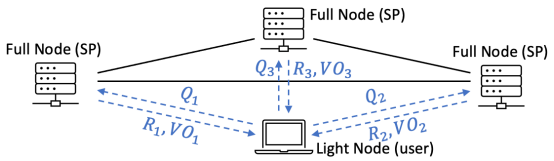

Figure 2: Authenticated Query Architecture.

Researchers have proposed several solutions for authenticated queries on blockchains, yet they still face three limitations.

(1) Dependence on trusted infrastructure. Approaches like vChain [38] and GCA^2-tree [43] use asymmetric accumulators [42] to verify query results via (non-)membership witnesses. However, managing public keys is impractical as their size scales with the maximum attribute value (*e.g.*, 2^{256} for 256-bit hashes). Alternatives, such as trusted oracles or specialized hardware, compromise blockchain decentralization and pose deployment challenges [34].

(2) Lack of aggregation support in predicate query solutions. Many frameworks, such as vChain [38], vChain+[34], MELTree[23], and SEBDB [44], support filtering data based on specific predicates (*e.g.*, boolean, range, etc.) but lack aggregation capabilities.

(3) Limited predicate support in aggregation query solutions. Solutions focused on aggregation provide limited predicate handling. For example, the sliding-window scheme [31] supports only temporal predicates at the block level, while GCA^2-tree [43] handles numerical predicates but not boolean queries on textual data.

In summary, authenticated aggregate queries with boolean and range predicates on blockchains face three fundamental challenges:

(1) Trustless Architecture: Preserving blockchain's trustless nature is challenging, as it requires eliminating reliance on trusted intermediaries without compromising efficiency or integrity.

(2) Comprehensive Aggregation: Supporting diverse aggregations of distributed blockchain data is costly, requiring significant computation and resources to process global summaries efficiently.

(3) Flexible Predicates: Providing boolean and numerical range predicates in queries requires efficient data indexing and filtering.

In this paper, we propose a trustless scheme for authenticated aggregate queries with boolean range predicates.

To address the challenge (1), we design *Merge Bloom Filter* (MBF), a symmetric accumulator based on *Bloom filters*, which (a) eliminates the trusted oracle in asymmetric accumulators, and (b) bounds the VO space overhead caused by false positive memberships.
To address the challenge (2), we develop an authenticated framework for aggregate queries, starting with MAX/MIN operations and extending to COUNT, COUNT DISTINCT, SUM, MEAN, and Top-k queries via sketch techniques.
To address the challenge (3), we propose the *Merkle Bloom Filter Tree* (MBFT), an ADS combining MHT and Bloom filter to support boolean keyword and range predicates over both numerical and set-valued attributes. Additionally, we introduce two optimizations to handle temporal predicates in time-window queries efficiently.

Our main contributions are summarized as follows:

- We design the *Merge Bloom Filter* (MBF), an efficient accumulator for authenticated queries that reduces the space overhead in the construction of ADS and the verification of results.
- We propose the *Merkle Bloom Filter Tree* (MBFT), an ADS scheme supporting various aggregate operations over time windows with boolean keyword and range predicates.
- We conduct experiments to validate the efficiency and effectiveness of our solution, demonstrating up to 286× acceleration in query time compared to state-of-the-art authenticated solutions.

The paper is organized as follows. Sec. 2 reviews related work and preliminaries. Sec. 3 defines the problem. Sec. 4 presents MBFT and MBF. We demonstrate the authenticated MAX query in Sec. 5 and extend to other aggregations in Sec. 6. Sec. 7 provides security anaylsis, and Sec. 8 reports experimental results. We conclude in Sec. 9.

2 RELATED WORK AND PRELIMINARIES

This section reviews related works on blockchain queries and cryptographic accumulators, followed by necessary preliminaries.

2.1 Blockchain Query

Hybrid Blockchain Database Queries. Existing blockchains lack native query support, leading to solutions incorporating external databases to build hybrid query engines [14]. BigchainDB [15] uses MongoDB for storage and Tendermint for consensus, while EtherQL [24] adds a query layer on top of Ethereum with MongoDB as data storage as well. When external users query these solutions, however, they have to fully trust the service provider to return correct results in the absence of independent verification, and thus unable to ensure integrity [34, 38].

Authenticated Queries on Blockchains. Another class of work enables authenticated queries directly on blockchains using authenticated data structures (ADS), enabling verifiable query execution.
Predicate Queries. Several solutions focus on authenticated predicate queries. SEBDB [44] employs the Merkle B-tree [21] for select-project-join queries, while vChain [38] and vChain+ [34] use multiset accumulators for boolean range queries. MELTree [23] and MP-DAG [22] support authenticated graph queries. However, these solutions lack comprehensive support for aggregation queries.
Aggregate Queries. Other approaches focus on authenticated aggregate queries. The Aggregate B-tree [31] supports sliding-window aggregation, while GCA^2-tree [43] provides generic aggregation capabilities. However, both systems have limited predicate support, *i.e.*, the Aggregate B-tree only supports block-level time predicates, and the GCA^2-tree only supports numerical range predicates.

Additionally, ADS solutions often rely on trusted oracles or hardware. For instance, vChain [38] and GCA^2-tree [43] require trusted oracles for public key generation and management, while multi-chain solutions like V^{2}FS [33] depend on trusted hardware (e.g., Intel SGX) through DCert [19] for data certification.

2.2 Cryptographic Accumulator

Merkle Hash Tree (MHT) and Variants. MHT is widely used in blockchains as the membership verification ADS [26, 36]. As shown in Fig. 3, each MHT leaf node corresponds to a transaction hash, and internal nodes are created by hashing the concatenation of child nodes. The root hash (*i.e.*, h_1) is stored in the block header. For example, to verify the membership of a transaction tx_1, the VO $\{h_5, h_3\}$ is returned as a proof. Then, the correctness can be verified by reconstructing the root hash h'_1 and comparing it with h_1.

Moreover, there are various MHT variants. Merkle Patricia Trie (MPT), used in Ethereum [36], efficiently manages the world state by prefix matching, while SEBDB [21] uses the Merkle B-tree for index authentication. Some systems combine MHT with asymmetric cryptographic accumulators to enable more expressive queries [38, 43], though these require trusted oracles for key management.

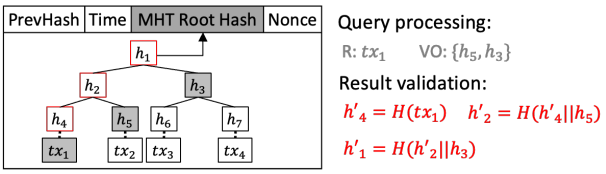

Figure 3: Merkle Hash Tree in Block.

Bloom Filter and Variants. Bloom filter (BF) [7], a space-efficient probabilistic symmetric accumulator, can also perform membership testing in static datasets with an acceptable rate of false positives (see Sec. 2.3). To support dynamic datasets, BF variants are proposed. The Dynamic Bloom Filter (DBF) [17] consists of multiple homogeneous Bloom filters, where one active filter inserts new elements, and once full, another empty filter is activated. The Scalable Bloom Filter (SBF) [2] consists of multiple heterogeneous Bloom filters, while the Dynamic Bloom Filter Array (DBA) [35] organizes Bloom filters into groups. These variants aim to handle dynamic sets and increase the capacity of the Bloom filter structure.

2.3 Preliminaries

Collision-resistant Hash Functions. A hash function H takes a variable-length input x and generates a fixed-length output $y = H(x)$. H is collision-resistant if the probability of finding two distinct inputs x_1 and x_2 such that $H(x_1) = H(x_2)$ is negligible. Typical collision-resistant examples are MurmurHash and SHA.

Bloom Filter. A Bloom filter (BF) uses an m-length bit vector, initialized to all 0, to represent a set X of n items. To insert an item $x \in X$, it maps x to k random positions $\{h_1(x), \ldots, h_k(x)\}$ where $h_i(x) = (H_i(x) \bmod m)$ via k independent hash functions and set these positions to 1. To check x's membership in X, BF computes k hashes $h_i(x)$ and returns true if all $h_i(x)$ bits are set to 1, false, otherwise.

Note that BF has a false positive rate ϵ due to hash collisions, which can be quantized as follows. Assume each $h_i(x)$ is evenly distributed over $[0, m-1]$ with the probability $\frac{1}{m}$ in each position. Given an m-bits BF with n items inside, the probability that one bit remains 0 is $(1 - \frac{1}{m})^{nk}$. Thus, the false positive rate ϵ follows:

$$\epsilon = [1 - (1 - \frac{1}{m})^{nk}]^k \approx (1 - e^{-kn/m})^k. \quad (1)$$

According to Eq. (1), when $k = \frac{m}{n} \ln 2$, ϵ reaches the minimum. Thus, the optimal m is derived as $m = -\frac{\ln \epsilon}{(\ln 2)^2} n$.

3 PROBLEM FORMULATION

This section formulates the problem by introducing the system and data models as well as the targeted queries and security goals.

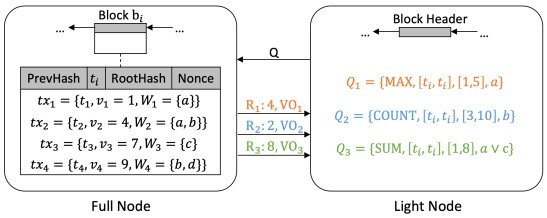

Figure 4: System Model.

System Model. We define our system model with two roles: (1) the service provider (SP), which stores the outsourced data and maintains the ADS to respond to user queries, and (2) the user, who requests aggregate queries from the SP. Specifically, we consider the following scenario: (1) The SP is untrusted, where the query result may be incorrect; (2) The user is resource-limited, retaining only the ADS digest and corresponding VO for query result verification. Typical applications are outsourced database [41], cloud services [37], edge computing [12], and blockchain [38].

This work focuses on the blockchain scenario. As illustrated in Fig. 4, the full node serves as the SP, maintaining blockchain data and the ADS, while the light node acts as the user, querying the full node and storing block headers for validation. Additionally, we aim to design an ADS that can be integrated into existing systems like Bitcoin and Ethereum by either replacing the native MHT root or appending the digest of our proposed ADS to the block header.

Data Model. We model the data as a sequence of append-only blocks, where each block b_i contains a list of transactions, *i.e.*, $b_i = (tx_1, \ldots, tx_n)$. Each transaction tx_j is modeled as $tx_j = \langle t_j, v_j, W_j \rangle$ where t_j, v_j, and W_j represent the timestamp, the numerical value, and the keyword set involved in this transaction, respectively.

Specifically, this data model applies to both UTXO-based and account-based blockchains, where v_j and W_j represent the transfer amount and textual properties (*e.g.*, user address) in a transaction, respectively. Moreover, smart contract transactions in the account-based blockchain may involve multiple numerical attributes. Following [38], we convert each numerical value into a binary prefix set and represent them in W_j. The range predicate on the numerical attribute v_j can be viewed as the minimum set of binary prefixes covering the range, and thus converted to Boolean keyword predicates on W_j. For instance, $\{1*, 10*, 100\}$ is the binary prefix set for the integer 4 (100 in binary), and the range predicate $[0, 4]$ can be transformed to the boolean predicate as $0* \vee 100$ (details see [38]).

Query Definition. We target on the aggregate query defined as $Q = \{\text{aggr}, [t_s, t_e], [\alpha, \beta], \Upsilon\}$, where (1) aggr is the aggregation operator including MAX, MIN, COUNT, COUNT DISTINCT, SUM, Top-k and MEAN (*i.e.*, $\frac{\text{SUM}}{\text{COUNT}}$); (2) $[t_s, t_e]$ is the temporal range predicate; (3) $[\alpha, \beta]$ is the numerical range predicate; and (4) Υ is the boolean function over the keyword set.

For example, Q_3 in Example 1.1 is represented as Q_3={COUNT, [2023-01-01, 2023-12-31], [1000, 5000], sender : 0x2ce7 $\wedge$ receiver : 0x3db6}, counting the retailer (0x2ce7) purchase records (value between 1000 and 5000) with the supplier (0x3db6) in year 2023.

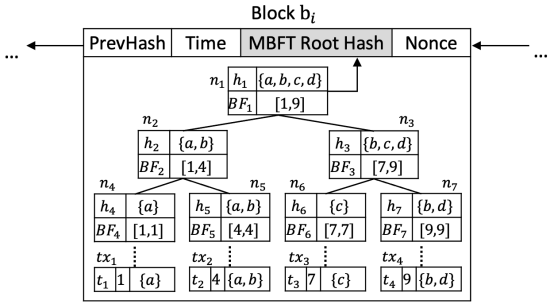

Figure 5: Example of Merkle Bloom Filter Tree.

Threat Model and Security Goal. We assume full nodes may be untrustworthy and could return tampered or incomplete VO and result, *e.g.*, a malicious attacker might exploit BF's false positives to produce fraudulent results that incorrectly match BF's keyword predicates. Meanwhile, we presume light nodes correctly follow the query scheme and maintain authentic block headers from the blockchain. To ensure data integrity, the full node generates the VO alongside the query result, and the light node can then verify both the *soundness* and *completeness* of the query result.

- **Soundness**. The returned query result is correct and not tampered with. It should satisfy all the selection predicates.
- **Completeness**. All qualified transactions are contained in the returned query result, and no valid data is missing.

4 BASIC SOLUTION

This section first introduces Merkle Bloom Filter Tree (MBFT), a novel ADS scheme, and its construction method (Sec. 4.1). Then, we propose an accumulator, Merge Bloom Filter (MBF), to optimize the verification object (VO) size of MBFT (Sec. 4.2).

4.1 MBFT Construction

Merkle Bloom Filter Tree. MBFT is built on the transaction list sorted by the numerical attribute. Fig. 5 illustrates an example MBFT with four transactions. Each tree node n_i contains a Bloom filter BF_i that accumulates keywords appearing in its subtree to support efficient tree-based index querying. Notably, the query complexity in Bloom filters is constant in k hash operations. It also provides a controllable false positive rate and accurate non-membership verification, ensuring the completeness of query results.

MBFT **node**. Each node n_i in MBFT consists of four components: (1) hash value h_i, an identifier of the node, where the tree root hash is stored in the block header, (2) keyword set W_i, containing keywords involved in the sub-tree rooted at node n_i, (3) Bloom filter BF_i, serving as an accumulator of the keyword set W_i, and (4) value range $[l_i, u_i]$, representing the lower (l_i) and upper (u_i) bounds of the transaction numerical value attribute involved in the sub-tree rooted at node n_i. The MBFT node is formally defined as:

*Definition 4.1 (*MBFT *Node).* Given an MBFT node n_i, we denote its left and right child as n_l and n_r, respectively. n_i contains the following four components, which can be computed as:

(1) $h_i = H(h_l \| h_r \| BF_i \| l_i \| u_i)$, where $\|$ compute the concatenation

(2) $W_i = W_l \cup W_r$

(3) $BF_i = bf(W_i)$ where $bf(\cdot)$ constructs a Bloom filter

(4) $[l_i, u_i]$ where $l_i = \min(l_l, l_r)$ and $u_i = \max(u_l, u_r)$

Algorithm 1: MBFT Construction.

```
1  Function MBFTConstruction(b):
       Input: Block b
       Output: MBFT root node
2      Initialize an empty node array Leaves and sort b into b';
3      for each transaction tx_i = ⟨t_i, v_i, W_i⟩ ∈ b' do
4          Compute BF_i, l_i, u_i and h_i;
5          Leaves.push(node(h_i, W_i, BF_i, l_i, r_i));
6      return MBFTMerge(Leaves);
7  Function MBFTMerge(Child):
       Input: Child node array Child
       Output: Parent node array Parent
8      if Child.size() is 1 then return Child;
9      if Child.size() is odd then Child.push(Child.tail());
10     for i ← 0 to Child.size()/2 do
11         ⟨h_l, W_l, BF_l, l_l, r_l⟩ ← Child [2 * i];
12         ⟨h_r, W_r, BF_r, l_r, r_r⟩ ← Child [2 * i + 1];
13         Compute W_i, BF_i, l_i, u_i and h_i;
14         Parent [i] = node(h_i, W_i, BF_i, l_i, r_i);
15     return MBFTMerge(Parents);
```

The MBFT construction algorithm is shown in Algo. 1. First, a sorted list b' is constructed according to the numerical value (line 2). For each transaction $tx_i \in b'$, we create a corresponding leaf node in MBFT (lines 3-5). Then we generate the MBFT in a bottom-up fashion until reaching the root node (line 8). Since the MBFT is a binary tree, the tail node should be computed twice for the odd number of nodes (line 9). Finally, parent nodes are generated by recursively merging each pair of child nodes (lines 10-15).

4.2 Merge Bloom Filter

Storage Optimization with Variable-Size Bloom Filters. We have derived the optimal size for the BF in MBFT root with n transactions as $-\frac{\ln \epsilon}{(\ln 2)^2} n$ in Sec. 2.3. However, this size is redundant for internal and leaf filters, which only store subsets of the transactions, leading to wasted storage. Consider the following example.

Example 4.2. Given a block with 100 transactions, each containing 2 addresses as keywords, and a target false positive rate $\epsilon = 1\%$, the optimal BF size in the MBFT root node is $-\frac{2 \cdot 100 \cdot \ln \epsilon}{(\ln 2)^2} \approx 240$B. However, the leaf node, which stores only 2 transactions with 4 keywords, requires only 4B of storage. Thus, using the same size for all BF in the MBFT leads to significant storage waste. □

A natural way to optimize storage is to use BFs of varying sizes in different nodes. However, this complicates merging child BFs to construct a parent BF, as merging BFs with different sizes requires rebuilding from scratch by reinserting all involved keywords. Although full nodes can handle this during ADS construction, a light node cannot reconstruct such a structure during VO verification, as they lack access to the underlying keyword set.

Merge Bloom Filter. To strike a balance, we propose the *Merge Bloom Filter* (MBF), a novel accumulator that allows merging two MBFs without knowledge of the underlying data items. This property of MBF enables efficient space utilization in MBFT formation, as it allows each MBF to dynamically adjust its size based on the number of accumulated elements, such that the MBF at deeper tree

levels has substantially smaller sizes. Specifically, there are three key components in the MBF: *dynamic hash*, *2-bit opcode*, and *hint*.

(1) Dynamic Hash. To support dynamic scaling, the positions of each item in the BF are adjusted when merging two MBFs. Recall that k hash functions determine an item's k positions, each computed as a hash value modulo the bit array size. When merging two BFs from child MBFT nodes, the parent node's attribute set and BF size approximately double. Based on this, we apply the dynamic hash technique to adjust item positions during construction.

Specifically, each MBFT node n_i with the set W_i stores each keyword x's overall position in binary form, *i.e.*, h_x=$H(x)$ mod m_{root}, where $H(x)$ is the hash of x and m_{root} is the root BF size. The position of x in BF_i is set by the last $len_i = \log\left(-\frac{\ln \epsilon}{(\ln 2)^2}|W_i|\right)$ bits of h_x, where BF_i has 2^{len_i} positions, scaling linearly with $|W_i|$ rather than the total keyword count across all transactions in the block.

(2) 2-bit Opcode. When users receive the VO, *i.e.*, MBFT branches with MBFs in each node, they cannot directly reconstruct the parent BF for verification without access to the complete keyword set. Therefore, we introduce 2-bit opcodes in MBF to facilitate merging. Specifically, MBF employs 2 bits to represent four possible states for each position: *empty* (00), *occupy and stay* (01), *occupy and move* (10), and *occupy and overlap* (11) constructed as follows:

- **Empty (00)**: The position is unoccupied.
- **Occupy and stay (01)**: The position is occupied by at least one item x, and the $(len_i + 1)$-th bit of h_x is 0.
- **Occupy and move (10)**: The position is occupied by at least one item x, and the $(len_i + 1)$-th bit of h_x is 1.
- **Occupy and overlap (11)**: The position is occupied by at least two items x_1, x_2, where the (len_i+1)-th bits of their h_{x_1} and h_{x_2} are 0 and 1, respectively.

The merging of two MBFs is first performed by the bitwise OR operation on the two child MBFs opcodes. As each opcode at position $p \in [0, len_i)$ indicates the bit state in the parent BF, then we transform it to a standard BF where **00** sets corresponding bits to 0, **01** (resp. **10**) sets the p-th (resp. $(len_i + p)$-th) to 1, and **11** sets both p-th and $(len_i + p)$-th bits to 1.

(3) Hint. After merging two MBFs, the resulting BF is a standard Bloom filter and lacks the 2-bit opcode information from the MBF at each position. Therefore, a hint is embedded in the VO to enable the transformation of the merged BF into an MBF. Specifically, each MBF node n_i involved in the MBFT VO path is accompanied by a hint, which is generated by observing the (len_i+2)-th bit of the h_x in each MBF where 0 (resp. 1) indicates the hint is **01** (resp. **10**). Then replace each non-zero bit in the merged BF with the corresponding 2-bit hint and each zero bit with 00 to obtain a valid MBF.

Algo. 2 illustrates the MBF reconstruction (*i.e.*, merging) process. First, two child MBFs are merged using a bitwise OR operation, creating a temporary MBF in the parent node that contains data from both child branches (line 2). Next, the 2-bit opcodes in the temporary MBF are converted into standard bits in the parent node's Bloom filter, which has the same size but twice the position capacity (lines 4-7). Finally, the parent node's MBF is reconstructed by traversing the parent BF's non-zero bits and assigning the corresponding opcodes from the hint at the same positions (lines 8-12).

The following example illustrates MBF and its merging process.

Algorithm 2: MBF Reconstruction.

```
1  Function MBFReconstruction(MBF_l, MBF_r, Hint):
       Input: Two child MBFs: MBF_l, MBF_r, and hint Hint
       Output: parent MBF MBF_p
2      MBF_c ← bitwiseOR(MBF_l, MBF_r);
3      BF_p←zeros(MBF_c.size()); MBF_p←zeros(MBF_c.size()∗2);
4      for i ← 0 to MBF_c.size()/2 do
5          OP ← ⟨MBF_c[2 ∗ i], MBF_c[2 ∗ i + 1]⟩;
6          if OP[1] is 1 then BF[i] ← 1;
7          if OP[0] is 1 then BF[i + MBF_c.size()/2] ← 1;
8      for i ← 0 to BF_p.size() do
9          if BF_p[i] is 1 then
10             OP ← Hint.pop();
11             MBF_p[2 ∗ i]←OP[1]; MBF_p[2 ∗ i + 1]←OP[0];
12     return MBF_p;
```

Example 4.3. Suppose an MBFT accumulates $|W| = 8$ keywords with an error rate $\epsilon = 0.38$, resulting in an optimal root BF size of $m_{\text{root}} = -\frac{\ln \epsilon}{(\ln 2)^2}|W| = 16$ positions. Consider leaf nodes MBF_l and MBF_r, containing keywords $W_l = \{w_1, w_2\}$ and $W_r = \{w_3, w_4\}$, respectively, with optimal BF sizes $m_l = m_r = -\frac{\ln \epsilon}{(\ln 2)^2}|W_l| = 4$ positions. Their hash values $h_x = H(x) \mod m_{\text{root}}$ are $\{6, 10, 13, 3\}$ in decimal form, with the binary representations shown in Fig. 6.

For keywords in W_l and W_r, we (a) determine their positions in MBF_l and MBF_r using the dynamic hash with $len_l = len_r = \log m_l = 2$, *i.e.*, the last 2 bits of h_x, yielding positions $\{10, 10, 01, 11\}$ for $w_1 \sim w_4$ and $\{2, 2, 1, 3\}$ in decimal; and (b) convert each position into a 2-bit opcode. In MBF_l, both w_1 and w_2 occupy position 2, with the $len_l + 1 = 3$rd bit of h_{x_1} and h_{x_2} being 1 and 0, respectively, yielding opcode **11** at position 2. In MBF_r, w_3 and w_4 occupy positions 1 and 3, with the 3rd bit of h_{x_3} and h_{x_4} being 1 and 0, respectively, yielding opcodes **10** and **01** for positions 1 and 3.

To merge MBF_l and MBF_r, we (a) compute the bitwise OR of the opcodes in MBF_l and MBF_r; (b) transfer the resulting opcodes to positions in the standard BF, with the optimal size $m = -\frac{\ln \epsilon}{(\ln 2)^2}(|W_l| + |W_r|) = 8$, *i.e.*, opcode **10** at position 1 sets the $len_i + 1 = 5$th bit to 1, **11** at position 2 sets bits at 2nd and $len_l + 2 = 6$th to 1, and **01** at position 3 sets the 3rd bit to 1; and (c) convert the 8 positions into the 2-bit representation by replacing 0 with 00 and 1 with the corresponding hint, which is identified as $\{01, 10, 10, 01\}$ based on the last $len_i + 2 = 4$th bit of each h_x. □

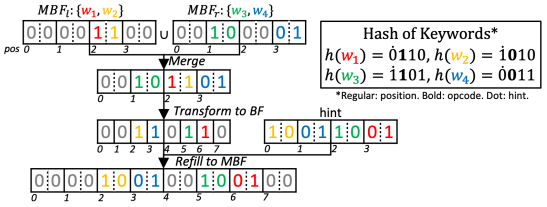

Figure 6: Example of Merge Bloom Filter.

Correctness Proof. Now we give the correctness proof of Algo. 2.

Theorem 4.4. *Algo. 2 can correctly reproduce the parent* MBF.

Proof. For each item x, (1) in child MBF_c (size 2^{len_c}), x has position $pos_c^x = h(x) \mod 2^{len_c}$ and opcode op_c^x=**01** (resp. **10**) if $h(x)$'s len_c+1-th bit is 0 (resp. 1); (2) in parent MBF_p (size $2^{len_p} =$

2^{len_c+1}), similarly, x has position $pos_p^x = h(x) \bmod 2^{len_p}$ with opcode op_p^x dependent on len_p+1-th bit; (3) hint $hint_c^x$ dependent on $h(x)$'s len_c+2-th bit. We prove that Algo. 2 outputs x's reconstructed $pos_p'^x$ and $op_p'^x$ in MBF_p equal to pos_p^x and op_p^x from three phases:

(1) *Merge.* After bitwise OR merging child MBFs, pos_c^x remains unchanged, while the opcode at pos_c^x may become **11** due to overlaps. However, as **11** covers operations from both **01** and **10**, op_c^x's operation information is preserved intact.

(2) *Transform to BF.* (a) If op_c^x=**01** (len_c+1-th bit is 0), $pos_p'^x$ remains at pos_c^x=$h(x)$ mod 2^{len_c}=$h(x)$ mod 2^{len_c+1}, thus $pos_p'^x = pos_p^x$. (b) If op_c^x=**10** (len_c+1-th bit is 1), $pos_p'^x$ moves to 2^{len_c}+pos_c^x=2^{len_c}+($h(x)$ mod 2^{len_c})=$h(x)$ mod 2^{len_c+1}, thus $pos_p'^x = pos_p^x$.

(3) *Refill to* MBF. Refill $hint^x$ to reconstruct $op_p'^x$, which depends on $len_c + 2 = len_p + 1$-th bit, thus $op_p'^x = op_p^x$.

Thus, Algo. 2 correctly reconstructs each item in MBF_p. □

Complexity Analysis. Finally, we analyze the time complexity for Algo. 2 and the storage cost for MBFT equipped with MBF.

Merge time complexity. Given two MBFs with n items in each, Algo. 2 consists of three steps: (1) computing bitwise OR on two MBFs vectors in $O(n)$ time; (2) converting 2-bit opcodes to standard one-bit BF position in $O(n)$ time; (3) refilling each one-bit position to a 2-bit MBF opcode via the hint in $O(n)$ time. The worst case occurs when all opcodes are **11** in step 2, which doubles the conversion time compared with opcodes **01** and **10**. Nevertheless, the overall time complexity remains $O(n)$.

Additionally, as the algorithm involves only linear scans and bitwise operations, it is well-suited for multithreading parallelization. By splitting the MBF into k chunks, each performing bit operations independently, the merge time complexity is reduced to $O(n/k)$.

Storage complexity. Given n items, the MBFT with standard BFs requires $O(n^2 + n \log n)$ space for BF vectors and keyword sets. Our proposed MBF reduces the MBFT storage cost to $O(n + n \log n)$. Moreover, the root MBF can be replaced with standard BF to further save space, as it doesn't need to carry extra opcode information.

5 AUTHENTICATED QUERY PROCESSING

This section explains how to process the authenticated MAX query in a single block using MBFT (Sec. 5.1). Due to the false positive rate in MBF, the search path may encounter an *astray*, thus we analyse the additional cost caused by astray (Sec. 5.2). Finally, we extend the method to support time window queries (Sec. 5.3).

5.1 Single Block Query

For the authenticated MAX query $Q = \{MAX, [t, t], [\alpha, \beta], \Upsilon\}$ over the block b timestamped at t with n transactions, its result is (1) processed by the full node and (2) verified by the light node. Specifically, the full node first utilizes the index in ADS to search for the target result and constructs the VO accordingly. Then, the light node verifies the result by reconstructing the MBFT root hash from the VO and comparing it with the one stored in the block header. The detailed procedure is illustrated below.

Authenticated Query Processing. The full node performs the query by traversing the MBFT from the root to the leaf containing the qualified transactions. For each node n_i on the search path, the full node accesses its MBF and value range $[l_i, u_i]$ to filter both the keywords and numerical attributes of the transactions.

- For a single keyword query $w \in \Upsilon$ over the node's keyword set W_i, it takes constant time to process on MBF_i with k hash operations and array indexing, regardless of $|W_i|$.
- For the boolean keyword predicate Υ, set operations in Υ are directly applicable to the MBF. For example, $\Upsilon = w_1 \wedge w_2$ passes if MBF_i contains both w_1 and w_2.
- For the value predicate $[\alpha, \beta]$, it is checked against the range $[l_i, u_i]$ in node n_i, where overlap indicates qualification.

Once both the boolean keyword and numerical predicates are satisfied, the full node continues along the search path, passing node n_i. For the MAX query, the full node visits the right child first, as it holds a larger value attribute; for the MIN query, it visits the left child first. Moreover, if node n_i fails to satisfy the selection predicates, the full node adds its hash value h_i, MBF MBF_i, and value range $[l_i, u_i]$ to the VO for non-membership proof, ensuring the integrity of the query result.

The detailed process of the single block query is illustrated in Algo. 3. The full node performs the MAX query with a depth-first tree search, using a stack maintaining unvisited nodes, and a flag indicating when the maximum node is reached (line 2). If the maximum result is found at a node, traversal stops for that branch, and its data is added to the VO for Merkle proof (line 5). As the search continues, we obtain the final result when a qualified leaf node is reached, then append the VO and mark the flag as true (lines 6-9). For internal nodes, the MBF hint is added to the VO, and its child nodes are pushed onto the stack (lines 10-13). For unqualified nodes, their data is added to the VO for non-membership proof (line 14).

Algorithm 3: Max Query Processing on Single Block.

```
1  Function MaxQuery(b, Q):
       Input: Block b, MAX query Q = {MAX, [t, t], [α, β], Υ}
       Output: Result R, verifiable object VO
2      stack.push(b's MBFT root); find ← false;
3      while stack not empty do
4          n_i ← stack.pop();
5          if find then VO.push(⟨h_i, BF_i, l_i, r_i⟩);
6          else if n_i matches Q then
7              if n_i is leaf then
8                  VO.push(⟨h_i, BF_i, l_i, r_i⟩);
9                  R ← n_i's transaction; find ← true;
10             else
11                 VO.push(n_i.hint());
12                 stack.push(n_i.leftChild);
13                 stack.push(n_i.rightChild);
14         else VO.push(⟨h_i, BF_i, l_i, r_i⟩);
15     return ⟨R, VO⟩;
```

We use the following example to explain the process.

Example 5.1. Consider the query $Q = \{MAX, [t_i, t_i], [1, 5], a \vee e\}$ on the block b_i shown in Fig. 7. The full node searches n_1, n_2 and n_5 to get the result $R = tx_2$, and generate the verifiable object $VO = \{\langle \mathsf{hint}_1 \rangle, \langle h_3, \mathsf{MBF}_3, 7, 9 \rangle, \langle \mathsf{hint}_2 \rangle, \langle h_5, \mathsf{MBF}_5, 4, 4 \rangle, \langle h_4, \mathsf{MBF}_4, 1, 1 \rangle\}$.

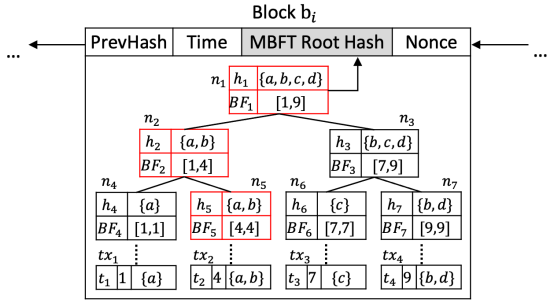

Figure 7: Example of Single Block Query.

Complexity analysis. The query processing detailed in Algo. 3 on a block with n transactions has a time complexity of $O(n)$. The tree search over the binary MBFT takes $\log n$ steps, with the hint generation cost at each step being linear in the size of the MBF. Thus, the overall time cost is $t = \sum_{i=0}^{\log n} O(2^i) = O(n)$. Notably, for each node in the VO, the keyword set is not used; only the MBF is returned due to its space efficiency and non-membership proof.

Result Verification. To ensure the data integrity of the result, the light node must verify both soundness and completeness with the help of the VO provided by the full node.

- **Soundness**. The light node reconstructs the MBFT root hash using the Merkle path in the VO and Algo. 2. Then compare the root with the one in the block header.
- **Completeness**. It verifies that objects larger than the result do not meet the predicates by ensuring that all right branches prove disqualification using the MBF and value range in the VO.

Algorithm 4: Max Query Verification on Single Block.

```
1  Function MaxVerify(Q, R, VO, h_root):
       Input: MAX query Q = {MAX, [t, t], [α, β], ϒ}, result R,
              verification object VO, root hash h_root
2      while VO not empty do
3          π ← VO.pop();
4          if π is R's node then Check R and π's validity;
5          else if π is right branch then
6              Check non-membership of ϒ in BF_sib;
7              Or check [α, β] ∩ [l_sib, u_sib] = ∅;
8          else if π is hint then
9              n_r ← stack.pop(); n_l ← stack.pop();
10             π ← reconstruct(n_l, n_r, hint);
11         stack.push(π);
12     Compare h_π and h_root;
```

The verification is illustrated in Algo. 4. The light node reconstructs the root hash bottom-up from each object π in the VO (lines 2-3). If π is the corresponding node of R, the light node verifies that R and π are qualified (line 4). Else, if it's on the right branch, the light node checks its failure to meet the predicates to ensure completeness (lines 5-7). Otherwise, if it's a hint, the light node reconstructs the parent node π from two child nodes poped from the stack(lines 8-10). In all cases, π is pushed to the stack (line 11) for MBF reconstruction later when hint occurs. Finally, the light node compares the reconstructed hash with the root hash in the header to complete the verification (line 12).

We use the following example to explain the process.

Example 5.2. Given R and VO from Example 5.1 in Fig. 7, the light node first verify R's keywords $\{a, b\}$ and value 4 matches predicates. Then, to reconstruct the MBFT root, n_2's MBF'_2 is rebuilt using MBF_4, MBF_5, and hint_2 by Algo. 2, its value range is updated to $[1, 4]$, and $h'_2 = H(h_4 \| h_5 \| \mathsf{MBF}'_2 \| 1 \| 4)$ is computed. Similarly, h'_1 is reconstructed and compared with h_1 in the header.

To verify that items larger than 4 (right branch) don't satisfy the predicates, the non-membership proof of $a \vee e$ in MBF_3 is checked, and the empty intersection between $[1, 4]$ and $[7, 9]$ is confirmed.

Complexity analysis. The result verification in Algo. 4 has a time complexity of $O(n)$. The reconstruction cost of each MBF is linear to its length, leading to an overall time cost of $t = \sum_{i=0}^{\log n} O(2^i) = O(n)$. Its bottom-up fashion restricts multithread acceleration, yet the parallel merge for each MBF reconstruction is compatible (Sec. 4.2) and can reduce the overall complexity to $O(n/k)$ as well. Similarly, the storage cost of the VO is also $O(n)$.

Remark. While Algo. 4 effectively verifies query results, we can further impose punishments for invalid results from malicious nodes. As it's orthogonal to our query scheme, various existing punishment mechanisms on blockchains can be integrated, such as stake slashing [30], reputation penalties [10], or economic sanctions through smart contracts [11]. These mechanisms can be implemented as modular extensions to our query scheme without compromising the protocol's overall security guarantees.

5.2 Astray Tree Analysis

An *astray tree* is a partial MBFT in which nodes incorrectly pass the BF query, while transactions in this branch fail to meet the keyword predicate, causing the query to follow an incorrect path.

The existence of an astray tree is caused by two main issues: (1) the inherent false positive rate in Bloom filters (and MBF), and (2) mismatches in conjunctive boolean predicates over accumulated keywords. For example, given the boolean predicate $\Upsilon = w_1 \wedge w_2$, a parent node with keyword set $W_p = \{w_1, w_2\}$ passes the predicate. However, neither of its child nodes n_l and n_r with keyword sets $W_l = \{w_1\}$ and $W_r = \{w_2\}$, satisfies Υ. As a result, the astray tree increases the VO size by introducing unnecessary branches.

We now analyze the extra storage cost introduced by astray trees and provide a theoretical proof. Let n denote the number of internal nodes and A_n represent the set of all astray trees with exactly n internal nodes. The size of A_n, denoted $f_n = |A_n|$, represents the frequency of all astray trees with n internal nodes. For example, Fig. 8 shows astray trees for $n \leq 3$, where $f_0 = f_1 = 1$, $f_2 = 2$, and $f_3 = 5$. The frequency f_n of astray trees has the following property.

Theorem 5.3. *Astray tree frequency f_n is a Catalan number C_n.*

Proof sketch. We prove the thereom by showing that (1) the frequency f_n for an astray tree with $n > 0$ internal nodes is determined by the combination of its child branches recursively as $f_{n+1} = \sum_{i=0}^{n} f_i \cdot f_{n-i}$; (2) when $n = 0$, it's a single node with $f_0 = 1$. Since these two conditions satisfy the recurrence relation of Catalan numbers, we can compute the frequency number f_n as $f_n = C_n = \frac{(2n)!}{n!(n+1)!}$. For a detailed proof, see Appendix A.1. □

With the formula for f_n, we now calculate the size of astray trees. Since the Bloom Filter dominates the VO space, we use the size of

the MBFs and corresponding hints to represent the storage cost of astray trees. On average, the hint is half the size of its corresponding MBF under optimal parameters. However, even for a fixed-shape astray tree, its size varies depending on the depth of its root node within the original MBFT. To isolate the impact of the astray tree structure, we analyze the ratio between the size of the astray tree and the size of its root node. For the j-th astray tree in A_n, we define its relative size as $r_{n,j}$. The total relative size of all astray trees in A_n is given by $r_n = \sum_{i=1}^{f_n} r_{n,j}$, which we illustrate through the following example.

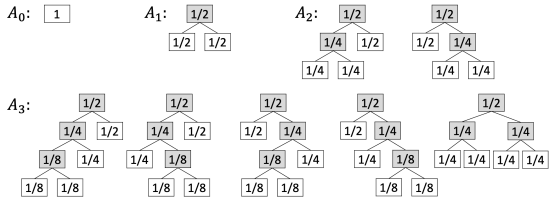

Figure 8: Example of Astray Trees.

Example 5.4. Fig. 8 illustrates relative sizes of astray trees with $n \leq 3$. When $n = 0$, the astray tree is a normal node in VO with $r_0 = r_{0,1} = 1$. When $n = 1$, its root hint, left child MBF and right child MBF are all half the size of its root MBF, thus $r_1 = r_{1,1} = \frac{3}{2}$. When $n = 2$, two astray trees in A_2 have same relative size $r_{2,1} = r_{2,2} = \frac{7}{4}$, and the sum r_2 is $\frac{7}{2}$. When $n = 3$, similarly, we can compute $r_{3,i} = \frac{15}{8}$ for $i \in \{1, 2, 3, 4\}$ and $r_{3,5} = 2$, thus the sum $r_3 = \frac{19}{2}$.

Then we prove how to compute the sum of relative size r_n, with the formal definition given below.

Theorem 5.5. *The sum of relative size is* $r_n = \frac{C_n + C_{n+1}}{2}$.

Proof sketch. We brief the proof through following steps: (1) for astray trees with $n > 0$ internal nodes, the sum of relatize size r_n is the sum of all combinations among its child branches, which can be summarized as $r_{n+1} = \frac{1}{2} f_{n+1} + \sum_{i=0}^{n} r_i f_{n-i}$; (2) since $f_n = C_n$ from Theorem 5.3, we have $f_{n+2} = \sum_{i=0}^{n} (f_i + f_{i+1}) f_{n-i}$; (3) then we can write $\frac{f_{n+1}+f_{n+2}}{2} = \frac{1}{2} f_{n+1} + \sum_{i=0}^{n} \frac{f_i + f_{i+1}}{2} f_{n-i}$, with the same expression pattern as r_{n+1} in step (1). Finally we conclude that $r_n = \frac{C_n + C_{n+1}}{2}$. For a detailed proof, see Appendix A.2. □

Given an MBF of size m, now we can compute its expected size m' with the extra space introduced by astray trees due to:

(1) The false positive rate ϵ_b from the Bloom filter. Since MBF can dynamically grow to maintain corresponding transactions at each node, a stable false positive rate is guaranteed as well. Therefore, we assume the MBF in each node has the same false positive rate ϵ_b.

(2) Mismatches in conjunctive predicates ϵ_c. For a predicate $\Upsilon = w_1 \wedge \cdots \wedge w_k$ with $k \geq 2$ keywords chosen from the keyword space $\mathcal{S}_w$ of size $|\mathcal{S}_w| = N$, consider two child nodes n_l and n_r each containing n keywords. Their parent node n_p contains $(2 - f_o)n$ keywords, where f_o is the overlap ratio of keywords. We define an indicator function $\infty_{k \leq f_o n}$ to check whether the required k conjunctive keywords fit within the overlap. The false positive rate for the parent node is then computed as: $\epsilon_c = 1 - \frac{2C_n^k - C_{f_o n}^k}{C_{(2-f_o)n}^k}$, and its limit value is: $\epsilon_c = 1 - 2\left(\frac{1}{2-f_o}\right)^k + \left(\frac{f_o}{2-f_o}\right)^k$, which depends on the predicate and keyword distribution.

Thus, the number of internal nodes n follows a Binomial distribution with probability ϵ. We can now compute the expected size m' of the MBF as below.

$$m' = m \sum_{i=0}^{\infty} r_i \epsilon^i (1 - \epsilon)^{i+1} = m \frac{2 - \epsilon}{2 - 2\epsilon} \tag{2}$$

For instance, with a false positive rate of $\epsilon = 5\%$, the extra storage cost introduced by astray trees is only 2.6% of the original MBF, which has a limited impact on our proposed ADS scheme.

In addition, to securely verify the queried result against the VO containing astray trees, both the query and verification algorithms must be augmented:

- **For query processing in Algo. 3**. If all reached children fail the query, we mark them as astray (lines 17-18) and stop the search on this branch.
- **For result verification in Algo. 4**. When we encounter a marked astray node, we perform an additional non-membership verification to ensure the correctness of the result.

5.3 Time Window Query

We then extend the single-block MAX query with MBFT to support the time window MAX query $Q = \{\text{MAX}, [t_i, t_{i+k}], [\alpha, \beta], \Upsilon\}$ across different blocks.

A naive approach is to traverse all blocks within the time period $[t_i, t_{i+k}]$ and query each block. Although each MBFT node contains a value range and MBF, allowing pruning of unnecessary queries without descending to the leaf level, in the worst case, where all blocks have qualified transactions, the time complexity for querying and validation is $O(k \log n)$, and the VO size is $O(k(2 - \frac{1}{n})m')$.

To improve the performance of time window queries, we propose two optimizations: (1) *value pruning*, which dynamically adjusts the query plan to reduce the search space, and (2) *multi-combination*, where multiple MBFTs are merged into one to reduce query time.

Value Pruning. We optimize the query plan by updating the range predicate $[\alpha, \beta]$ as the maximal result changes across blocks, and pruning nodes with smaller values for the MAX query ahead of time.

For example, after querying block b_j with timestamp $t_j \in [t_i, t_{i+k}]$ and obtaining a maximal result r_j in b_j, we update the query range predicate from $[\alpha, \beta]$ to $[r_j, \beta]$. Consequently, the next query on block b_{j+1} is less likely to satisfy the predicates, allowing an early stop and saving both query time and VO size.

For validation, light nodes can synchronize this update by following the temporal order of blocks. Assuming the maximal result is uniformly distributed over $[t_i, t_{i+k}]$, on average, blocks within $[t_{i+\frac{k}{2}}, t_{i+k}]$ will encounter an early stop to reduce the query cost.

Multi-combination. Furthermore, we propose the optimization, which periodically merges the MBFTs of multiple blocks into one larger MBFT, enabling the full node to query transactions across multiple blocks with a single tree search. This also improves verification performance by verifying non-membership in one step.

Specifically, the full node combines MBFTs from every τ blocks and appends the root hash of the combined MBFT to the last block in the cycle. It also maintains a sorted list of all transactions involved.

When a new block b_j within the combination cycle $[t_i, t_\tau]$ is mined, the sorted list is updated by merging it with the new block's sorted list, which takes $O((j-i)n)$ time, negligible compared to $O(n \log n)$ for sorting a single block.

Therefore, when the light node requests a time window query, it can be divided according to the combination cycle. If the window is too short to contain a full cycle, the query can be handled by traversing individual blocks with the query plan optimized by value pruning. For larger windows containing complete cycles, the full node can apply the multi-combination solution to blocks within the cycles and use the traversal solution for the remaining blocks.

6 EXTENSION TO OTHER AGGREGATIONS

In this section, we extend our authenticated MAX query to support other aggregate queries, including COUNT, COUNT DISTINCT, SUM, MEAN, and Top-k-related queries, noting that the MIN query is processed in a similar way to MAX, differing only in comparison.

Inspired by *sketch* techniques widely used in streaming databases and sensor networks, we leverage different sketches to answer aggregate queries with guaranteed approximation and constrained space cost [9, 13, 28]. By sacrificing some accuracy, we can avoid the high cost of asymmetric cryptographic accumulators [42] and provide more efficient aggregate sketches to light nodes based on the MAX query solution proposed earlier. We summarize the extended aggregations with supporting mechanisms in Table 1.

Aggregate	Supporting Mechanism
COUNT (DISTINCT)	AMS/Uniform Sketch, seeded with tx's unique hash (target attribute).
SUM, MEAN	Construct v multiple COUNT sketches and reserve the max one.
Top-k related	Invoke MAX query k times and update query predicate.

Table 1: Aggregate Extension Summary.

COUNT **and** COUNT DISTINCT **Query**. Given a COUNT query $Q = \{\text{COUNT}, [t_s, t_e], [\alpha, \beta], \Upsilon\}$, the full node returns the number of transactions that satisfy the given predicates. We extend our MAX query solution to securely answer COUNT queries by leveraging existing count sketches and approximate algorithms [28].

COUNT *Query Using AMS Sketch.* One way to achieve the authenticated COUNT query is to use the AMS sketch [3]. For each transaction tx_i, its sketch s_i is a random integer $x \in \mathbb{N}^+$ drawn with the probability of 2^{-x}. By finding the maximal sketch value s_m among all qualified transactions, we can estimate the count as 2^{s_m}.

To ensure correctness, the sketch s_i is generated by a pseudo-random number generator with a publicly-verifiable seed (*e.g.*, the transaction hash) [29] such that both full and light nodes can verify the validity of each sketch. With each transaction attached by a verifiable sketch, the full node can sort transactions by their sketch values and construct the MBFT as the ADS for the COUNT query.

To process a COUNT query on a single block, the full node traverses the MBFT and checks if the nodes satisfy the predicates. By first visiting the right child nodes, the search ensures obtaining the maximal sketch value s_m of the qualified transactions, and all transactions on its right fail the predicates. The COUNT is then estimated as 2^{s_m}, and the full node returns the result, along with the VO introduced in Sec. 5.1, which is verified by the light node.

To optimize with value pruning (Sec. 5.3), we attach an additional data field containing the maximum sketch value of the subtree rooted at each node. This is necessary because the numerical range does not align with the ascending order of sketch values. Moreover, the time window COUNT query can be optimized by the multi-combination MBFT in (Sec. 5.3) as well.

COUNT *Query Using Uniform Sketch.* Another way to achieve the COUNT query is to use the uniform sketch, where the sketch s_i for each transaction tx_i follows the uniform distribution $U([0, 1])$. Similar to the AMS sketch, we apply the MAX query on the sketch to identify the transaction with the maximal sketch value s_m and estimate the count as $\frac{1}{1-s_m}$. Additionally, it enables improving the estimation accuracy by finding the top-k maximal sketch and estimating the count as $\frac{k}{1-s_k}$, where s_k is the k-th largest value [5]. A detailed discussion of the top-k query will be provided later.

COUNT DISTINCT *Query.* Once the authenticated COUNT query is established, it can be easily extended to the COUNT DISTINCT query, which counts the number of unique values for a specific transaction attribute, such as the transaction amount or receiver address. To achieve this, we modify the random number generator's seed from the transaction hash to the selected attribute (*e.g.*, amount or receiver address), ensuring that transactions with the same attribute value share the same sketch value. Thus, the COUNT sketches can be directly used for distinct count aggregation.

SUM **and** MEAN **Query.** To extend the COUNT query to the SUM query, we treat each transaction tx_i with the numerical value v_i (assumed to be an integer) as a set of v_i distinct transactions, each with a unit value of 1. Consequently, summing the transaction values becomes counting the total number of such unit transactions. Specifically, for each unit transaction in the set of value v_i, we associate it with a count sketch $s_{i,j}$ for $j \le v_i$. By merging these sketches, we estimate the sum of transaction tx_i as $\max_{j \le v_i} s_{i,j}$. Note that blockchain transaction values are typically integers, as they correspond to the smallest unit of the cryptocurrency. Moreover, we can generate multiple sketches in one step to improve the efficiency [9, 27]. Finally, the MEAN query can be derived by dividing the SUM query result by the COUNT query result.

Top-k **Related Query.** To answer the Top-k query with value predicates $[\alpha, \beta]$, we can invoke the MAX query k times sequentially. After each MAX query, the maximum result v_m from the previous round is excluded by updating the upper bound of the value range to $[\alpha, v_m)$ for the next query.

For time window Top-k queries, the combination solution allows the full node to perform a single Top-k search over the combined MBFT, instead of querying each block individually. When the query spans multiple combination windows, we find the top k values $\{v_1, \ldots, v_k\}$ in descending order from the first window and update the value predicates into $(v_k, \beta]$ for subsequent MBFTs to reduce the search space. This refinement consistently updates the top k values and triggers more early stops, improving query efficiency.

Other Top-k-related queries can be supported by extending the Top-k query [28]. For example, in the Uniform Sample query, the full node generates a uniform sketch in $(0, 1)$ for each transaction and retrieves the top k sketches [27]. Based on this, the full node can compute the Median, Quantile, and k-th statistical moments [27].

7 SECURITY ANALYSIS

In this section, we perform a comprehensive security analysis.

False Positive Issue. In our threat model (see Sec. 3), a malicious full node might manipulate the VO and the result, particularly by exploiting the flaw of BF's false positives, *i.e.*, the BF incorrectly reports the set membership of an item x, while x is not a valid member. This enables the malicious node to produce a result falsely claiming that x passes the keyword query predicate.

Note that even if an attacker injects BF collision data to forge an invalid VO or result, the integrity checks (*i.e.*, root hash or transaction signature) will still fail. While the false positive might increase the VO size by involving mismatches of MBFT nodes with incorrect keywords, it does not compromise the soundness or completeness of the query result verification. We analyze as follows.

(1) BF false positive doesn't compromise completeness. False positives only result in incorrect membership proofs, not non-membership proofs. Therefore, the adversary cannot generate incomplete results by forging non-membership proofs.

(2) Soundness is ensured through secondary checks. Even if an incorrect MBFT leaf node passes the BF membership check, the user can perform a secondary predicate check on the returned query result. Thus, BF false positives only impact the VO size, which is bounded (see Sec. 5.2), and do not introduce security issues.

Overall Security. Next, we formally define our security goal and prove that our scheme is secure against any malicious attacks.

Definition 7.1. (Secure) Our query and verification algorithms are secure if for all probabilistic polynomial time adversaries, their probability to succeed in the following experiment is negligible:

- Construct an MBFT on the dataset D according to the construction algorithm Algo. 1 and send D to the adversary.
- Adversary outputs an aggregate query with boolean range predicates Q, a forged result R', and a forged proof VO'.

The adversary succeeds if the forged VO' and R' pass the verification while R' is different from the actual query result R.

The above definition demonstrates that the probability for a malicious full node to convince the light node with a fraud result is negligible. Below, we prove that our proposed verification algorithm satisfies such requirements.

Theorem 7.2. *Our query and verification algorithms are secure as defined in Def. 7.1, if the cryptographic hash function is collision-resistant, and the blockchain data is secure.*

Proof. We establish the proof in the following three steps.

(1) The adversary returns a forged $R' \notin D$. The integrity of the result can be verified by *(a)* checking R''s transaction hash and signature; *(b)* reconstructing the root hash based on the forged R' and VO, and comparing it with the root hash of dataset D from the blockchain header. If they match, it contradicts the statement that the hash function is collision-resistant.

(2) The adversary returns a result R' doesn't satisfy the boolean range predicate. It can be easily checked since the light node can check the qualification of the forged result by verifying the raw transaction's value and keywords.

(3) The adversary ignores the actual result R. In this case, the light node can verify the returned VO' by checking if each leaf node contains a non-membership proof, which is impossible for the actual VO with qualified transactions involved. The integrity of VO can be checked by reconstructing the root hash and compare against the on-chain hash. □

8 EXPERIMENTS

In this section, we evaluate the performance of MBFT with MBF for aggregate queries with boolean range predicates.

8.1 Experiment Setting

Real Dataset. We conduct experiments on the real dataset [34] extracted from Ethereum [36], containing around 680,000 transactions in 20,000 blocks. We represent each transaction in the form of $\langle id, timestamp, amount, \{addresses\}\rangle$, where id also serves as the seed for COUNT sketch verification (see Sec. 6), *{addresses}* contains the addresses of the sender and receiver.

Baselines and Metrics. We implemented following baseslines.

- MHT, the de facto ADS widely adopted in blockchains.
- MongoDB [25], the database utilized by BigchainDB [15] and EtherQL [24] for queries on full nodes.
- BigQuery [6], Google Cloud's serverless data warehouse, supporting third-party queries over the Ethereum dataset.
- vChain+ [34], a state-of-the-art authenticated query solution for boolean range predicates, accelerated by multithread. We manually aggregate matching transactions at the user side.
- GCA^2-tree [43], a verifiable aggregate query solution over numerical range predicates. It cannot support keyword predicates.
- MBFT under different modifications: (1) MBFT-BF, using standard BF instead of MBF; (2) MBFT-nil, using vanilla MBF; (3) MBFT-vp, enabling value pruning optimization (see Sec. 5.3); (4) MBFT-mc, enabling multi-combination optimization (see Sec. 5.3); (5) MBFT-all, enabling both value pruning and multi-combination optimizations.

We evaluate the ADS setup cost by measuring the construction time and ADS size per block. We evaluate the query performance by measuring the following three metrics: (1) the CPU time of the full node for query processing; (2) the CPU time of the light node for result verification; (3) the size of VO for transmission.

Parameters	Settings
False positive rate ϵ	0.01, 0.03, **0.05**, 0.07, 0.09
Time window	400, **800**, 1200, 1600, 2000
Block size	32, 64, **128**, 256, 512
Multi-combination cycle	1, 2, 4, **8**, 16
Aggregate types	MAX, **COUNT** (DISTINCT), SUM
Numerical selectivity	10%, 30%, **50%**, 70%, 90%
Number of keywords	1, **2**, 4, 8, 16

Table 2: Parameter Setting

Workloads and Settings. For each experiment, we generate 100 queries under the parameter settings in Table 2 where default values are in bold. Specifically, we focus on the query template $Q = \{aggr, [t_s, t_e], [\alpha, \beta], \Upsilon\}$ defined in Sec. 3 by varying: (1) the false positive rate to test its impact on the VO size of MBFT equipped; (2) the block size to compare ADS construction cost;

(3) the length of time windows to test the performance under two different boolean function: ∧- and ∨-connected boolean range predicates; (4) the number of keywords and the selectivity of numerical range to test the impact of query predicate selectivity; (5) the aggregate functions to measure their performance; (6) the combination cycle and block size to compare the scalability of different ADS.

Experimental Environment. We conduct experiments on a server with 96-core Intel(R) Xeon(R) Gold 6240R CPU, running on CentOS 7. The experiments are implemented in C++ with Murmurhash3 [1] as the hash function. All tests on MBFT are performed in a single thread and run 100 times, where the average results are reported.

8.2 Experimental Results

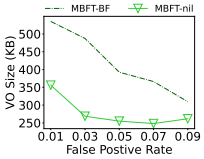

Figure 9: Varying ϵ

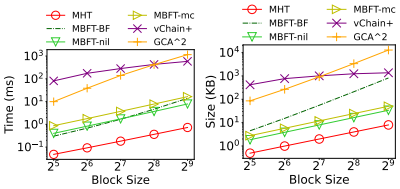

Figure 10: ADS Construction Cost

8.2.1 False Positive Rate. We first evaluate the impact of the false positive rate ϵ on the VO size.

Vary ϵ. As shown in Fig. 9, MBFT-nil reduces MBFT-BF's VO size by 32% on average. Increasing the false positive rate ϵ reduces the VO size by 26% for MBFT-nil and 42% for MBFT-BF. With MBF dynamically adjusting its size, MBFT-nil's child nodes can be smaller than those in MBFT-BF, resulting in a more compact VO. As ϵ increases, BF/MBF requires less space to represent the original set, further reducing VO size. Thus, although a larger ϵ may theoretically introduce more astray trees and incur additional space costs, this effect is bounded (see Sec. 5.2) and is empirically outweighed by the size savings from smaller BF/MBF structures.

8.2.2 ADS Setup Cost. We then compare the construction time and space cost of ADS across different schemes.

Vary Block Size. As shown in Fig. 10, MBFT-nil achieves up to 209× and 199× reductions in construction time and space cost compared to vChain+ and GCA^2-tree. Specifically, GCA^2-tree requires 140 ms for time and 919 KB for space to compute an expressive accumulator digest over the sorted transaction list. vChain+ takes 282 ms and 0.98 MB to construct ADSs for each sliding window and maintain the object registration index for transaction IDs. In contrast, MBFT only sorts the transactions and performs hash operations for each MBF, requiring 1.7 ms for time and 7.97 KB for space.

Furthermore, by employing the space-efficient MBF, MBFT achieves a 7× space reduction compared to MBFT-BF. While MBFT-mc with 8 combined blocks incurs a slightly higher setup time and space than MBFT-nil, averaging 3.7 ms and 11.85 KB, it is still significantly lower than vChain+ and GCA^2-tree. Finally, all methods incur the same storage overhead on light nodes.

8.2.3 Time Window Query Performance. Next, we evaluate the time window query performance under varying the time window size, predicate selectivity, and aggregate types.

Varying time window size. As shown in Fig. 11 and Fig. 12, MBFT (1) achieves query performance on par with BigQuery and MongoDB, and (2) significantly outperforms vChain+ and GCA^2-tree, with up to 286 × and 17,000× faster query time, respectively.

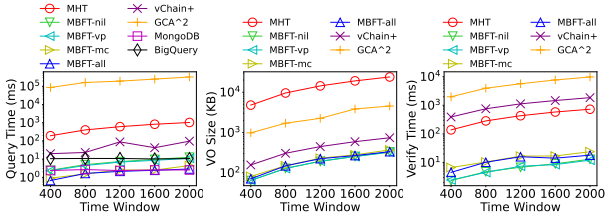

Figure 11: ∧-Connected Boolean Range Predicate Performance

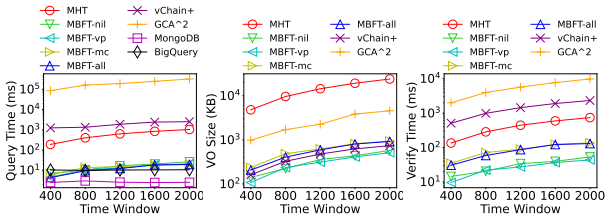

Figure 12: ∨-Connected Boolean Range Predicate Performance

Comparing with non-authenticated solutions. Under the ∨ boolean predicate, MongoDB achieves the fastest query time at 2.95 ms, due to advanced database features such as efficient indexing, outperforming MHT's 399 ms on full nodes. BigQuery also performs efficiently at 9.78 ms, slightly slower than MongoDB due to the need to collect all transactions before filtering. Despite the overhead of proof generation, MBFT also serves as an efficient boolean range index using lightweight hash operations, achieving 10.28 ms query time which is comparable to non-authenticated solutions.

Comparing with authenticated solutions. vChain+ and GCA^2-tree incur significantly higher query times of 1.33 s and 161 s, and verification times of 0.98 s and 3.95 s, respectively, due to the expensive bilinear pairing computations. vChain+ outperforms GCA^2-tree by leveraging a multi-window index design and enabling multithreading across all 96 CPU cores. MHT yields the largest VO size, as it must return each individual transaction for verification. In contrast, MBFT achieves the best performance by using simple BF queries to eliminate unqualified blocks and relying solely on hash operations for reconstruction. Furthermore, both MBFT-vp and MBFT-mc enhance query efficiency under both boolean predicates. However, MBFT-mc incurs longer verification time and a larger VO size than MBFT-nil, as it reconstructs a combined MBFT.

Varying predicate selectivity. As shown in Fig. 13 and Fig. 14, MBFT consistently outperforms MHT, vChain+, and GCA^2-tree. While MongoDB and BigQuery maintain stable performance due to database-level optimizations, vChain+'s query and verification times degrade by 1,800× and 54×, respectively, as the boolean function size increases, due to costly cryptographic intersection operations over high-dimensional keyword sets. In contrast, MBFT-nil exhibits a stable linear growth, *i.e.*, 8× in query time and 6.9× in verification time, since it only incurs lightweight bit operations.

Varying aggregate types. As shown in Fig. 15, all aggregate functions exhibit similar performance. MongoDB and BigQuery leverage database-level aggregation features, resulting in stable query

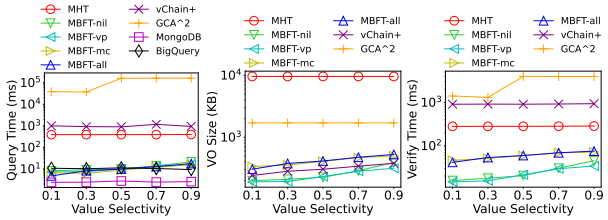

Figure 13: Impact of Numerical Range Selectivity

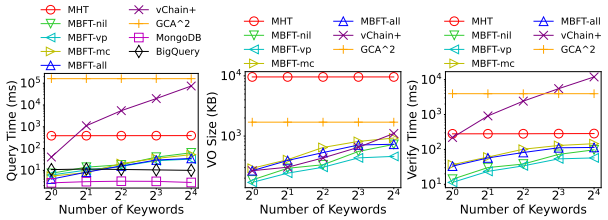

Figure 14: Impact of Boolean Function Size

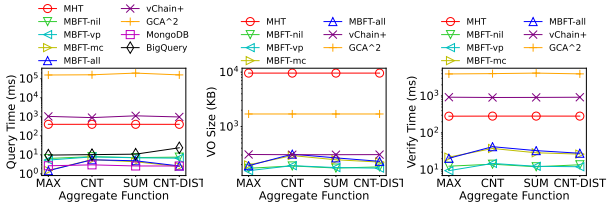

Figure 15: Impact of Aggregation Type

times. Since MHT and vChain+ do not support aggregation natively, aggregation is performed locally over the qualified transactions, where predicate query and verification dominate the overall performance rather than the aggregation computation. In contrast, GCA^2-tree applies expressive set accumulators for aggregation, achieving comparable results. MBFT utilizes sketches generated during ADS construction, which introduces no additional cost for query or verification. As all aggregate types follow the same execution workflow (see Sec. 6), they exhibit consistently stable performance. Compared to GCA^2-tree, MBFT-all has 59,341× faster query speed and MBFT-vp has 336× faster verification time on average.

8.2.4 Scalability. Finally, we evaluate the scalability of MBFT.

Varying block size. As shown in Fig. 16, MBFT's VO size and verification time grow linearly, consistent with its space and time complexity (see Sec. 5.1). Even at large scales, MBFT remains up to 26× faster than vChain+ and GCA^2-tree, as its MBF merge relies on efficient bit operations rather than costly cryptographic set operations. Moreover, it shows that the MBFT merge process does not become a performance bottleneck when scaling.

Varying combine cycle. As in Fig. 17, MBFT-mc and MBFT-all improve query performance for the ∧ boolean function by up to 1.6× with a large combination cycle. Compared to MBFT-nil, MBFT-mc achieves a 3.24× speedup, resulting in lower query time from the full node as the SP. While more combinations lead to longer verification times, the cost is still within acceptable limits

8.2.5 Summary of Experimental Results. We observe that (1) MBFT-nil reduces MBFT-BF's VO size by 32% on average and the increased false positive rate ϵ reduces VO size for both MBFT-nil and MBFT-BF; (2) MBFT-nil achieves the lowest ADS setup cost,

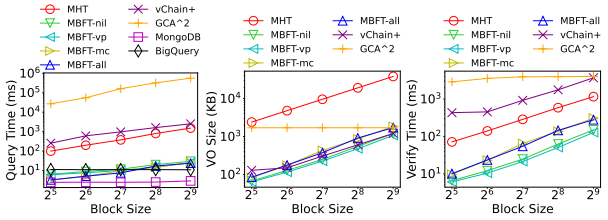

Figure 16: Impact of Block Size

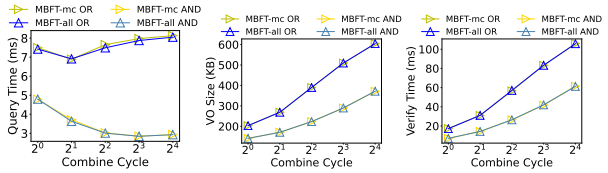

Figure 17: Impact of Combination Cycle

requiring only 1.7 ms and 7.97 KB, offering up to 209× and 199× reductions in time and space compared to vChain+ and GCA^2-tree; (3) MBFT matches non-authenticated databases in query speed (10.28 ms vs. 9.78 ms in BigQuery), while outperforming state-of-the-art authenticated solutions with up to 286× faster queries; and (4) MBFT maintains stable performance in varying block size, predicate selectivity, and aggregate types, and benefits further from combination and value pruning with moderate reconstruction cost.

9 CONCLUSION

In this paper, we propose Merkle Bloom Filter Tree (MBFT), an authenticated data structure for efficient aggregate queries under boolean and range predicates on blockchain. MBFT integrates Bloom filters and value ranges for flexible indexing. To further improve performance, we propose a space-efficient Merge Bloom Filter (MBF) to adapt to dynamic data volumes with value pruning and block combination optimizations. We use data sketches to support diverse aggregate operations. Experimental results confirm that our approach significantly reduces query time, proof size, and verification cost compared to existing solutions.

ACKNOWLEDGMENTS

This research is supported by the Research Grants Council (RGC) of Hong Kong under the Early Career Scheme (ECS) Project Number 25600624. Lei Chen's work is partially supported by National Key Research and Development Program of China Grant No. 2023YFF0725100, National Science Foundation of China (NSFC) under Grant No. U22B2060, Guangdong-Hong Kong Technology Innovation Joint Funding Scheme Project No. 2024A0505040012, the Hong Kong RGC GRF Project 16213620, RIF Project R6020-19, AOE Project AoE/E-603/18, Theme-based project TRS T41-603/20R, CRF Project C2004-21G, Key Areas Special Project of Guangdong Provincial Universities 2024ZDZX1006, Guangdong Province Science and Technology Plan Project 2023A0505030011, Guangzhou municipality big data intelligence key lab, 2023A03J0012, Hong Kong ITC ITF grants MHX/078/21 and PRP/004/22FX, Zhujiang scholar program 2021JC02X170, Microsoft Research Asia Collaborative Research Grant, HKUST-Webank joint research lab and 2023 HKUST Shenzhen-Hong Kong Collaborative Innovation Institute Green Sustainability Special Fundfrom Shui On Xintiandi and the InnoSpace GBA.

REFERENCES

[1] aappleby. 2025. Murmurhash3. https://github.com/aappleby/smhasher.

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. 2007. Scalable bloom filters. *Inform. Process. Lett.* 101, 6 (2007), 255–261.

[3] Noga Alon, Yossi Matias, and Mario Szegedy. 1999. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences* 58, 1 (1999), 137–147.

[4] Mohammad Javad Amiri, Chenyuan Wu, Divyakant Agrawal, Amr El Abbadi, Boon Thau Loo, and Mohammad Sadoghi. 2024. The bedrock of byzantine fault tolerance: A unified platform for {BFT} protocols analysis, implementation, and experimentation. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 371–400.

[5] Ziv Bar-Yossef, T S Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan. 2002. Counting distinct elements in a data stream. In *International Workshop on Randomization and Approximation Techniques in Computer Science*. Springer, 1–10.

[6] BigQuery. 2025. BigQuery official website. https://cloud.google.com/bigquery.

[7] Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[8] Luca Cabibbo and Riccardo Torlone. 1999. A framework for the investigation of aggregate functions in database queries. In *International Conference on Database Theory*. Springer, 383–397.

[9] Jeffrey Considine, Feifei Li, George Kollios, and John Byers. 2004. Approximate aggregation techniques for sensor databases. In *Proceedings. 20th International Conference on Data Engineering*. IEEE, 449–460.

[10] Marcela T de Oliveira, Lúcio HA Reis, Dianne SV Medeiros, Ricardo C Carrano, Sílvia D Olabarriaga, and Diogo MF Mattos. 2020. Blockchain reputation-based consensus: A scalable and resilient mechanism for distributed mistrusting applications. *Computer Networks* 179 (2020), 107367.

[11] Harsh Desai, Kevin Liu, Murat Kantarcioglu, and Lalana Kagal. 2018. Adjudicating violations in data sharing agreements using smart contracts. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 1553–1560.

[12] Chaosheng Feng, Keping Yu, Moayad Aloqaily, Mamoun Alazab, Zhihan Lv, and Shahid Mumtaz. 2020. Attribute-based encryption with parallel outsourced decryption for edge intelligent IoV. *IEEE Transactions on Vehicular Technology* 69, 11 (2020), 13784–13795.

[13] Philippe Flajolet and G Nigel Martin. 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences* 31, 2 (1985), 182–209.

[14] Zerui Ge, Dumitrel Loghin, Beng Chin Ooi, Pingcheng Ruan, and Tianwen Wang. 2022. Hybrid Blockchain Database Systems: Design and Performance. *VLDB Endowment* 15, 5 (2022), 1092–1104.

[15] BigchainDB GmbH. 2018. BigchainDB 2.0: the blockchain database.

[16] Stephane Grumbach, Maurizio Rafanelli, and Leonardo Tininini. 1999. Querying aggregate data. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 174–184.

[17] Deke Guo, Jie Wu, Honghui Chen, Ye Yuan, and Xueshan Luo. 2009. The dynamic bloom filters. *IEEE Transactions on Knowledge and Data Engineering* 22, 1 (2009), 120–133.

[18] Siyuan Han, Zihuan Xu, Yuxiang Zeng, and Lei Chen. 2019. Fluid: A blockchain based framework for crowdsourcing. In *Proceedings of the 2019 international conference on management of data*. 1921–1924.

[19] Yang Ji, Cheng Xu, Ce Zhang, and Jianliang Xu. 2022. DCert: towards secure, efficient, and versatile blockchain light clients. In *Proceedings of the 23rd ACM/IFIP International Middleware Conference*. 269–280.

[20] Kari Korpela, Jukka Hallikas, and Tomi Dahlberg. 2017. Digital supply chain transformation toward blockchain integration. In *proceedings of the 50th Hawaii international conference on system sciences*.

[21] Feifei Li, Marios Hadjieleftheriou, George Kollios, and Leonid Reyzin. 2006. Dynamic authenticated index structures for outsourced databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 121–132.

[22] Siyu Li, Zhiwei Zhang, Jiang Xiao, Meihui Zhang, Ye Yuan, and Guoren Wang. 2024. Authenticated Keyword Search on Large-Scale Graphs in Hybrid-Storage Blockchains. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1958–1971.

[23] Siyu Li, Zhiwei Zhang, Meihui Zhang, Ye Yuan, and Guoren Wang. 2024. Authenticated Subgraph Matching in Hybrid-Storage Blockchains. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1986–1998.

[24] Yang Li, Kai Zheng, Ying Yan, Qi Liu, and Xiaofang Zhou. 2017. EtherQL: a query layer for blockchain system. In *International Conference on Database Systems for Advanced Applications*. Springer, 556–567.

[25] MongoDB. [n.d.]. MongoDB official website. https://www.mongodb.com.

[26] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review* (2008), 21260.

[27] Suman Nath, Phillip B Gibbons, Srinivasan Seshan, and Zachary Anderson. 2008. Synopsis diffusion for robust aggregation in sensor networks. *ACM Transactions on Sensor Networks (TOSN)* 4, 2 (2008), 1–40.

[28] Suman Nath, Haifeng Yu, and Haowen Chan. 2009. Secure outsourced aggregation via one-way chains. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 31–44.

[29] Bartosz Przydatek, Dawn Song, and Adrian Perrig. 2003. SIA: Secure information aggregation in sensor networks. In *Proceedings of the 1st international conference on Embedded networked sensor systems*. 255–265.

[30] Tim Roughgarden. 2024. Keynote: Provable Slashing Guarantees. In *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing*. 2–2.

[31] Qifeng Shao, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2021. Trusted sliding-window aggregation over blockchains. In *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 257–265.

[32] Philip Treleaven, Richard Gendal Brown, and Danny Yang. 2017. Blockchain technology in finance. *Computer* 50, 9 (2017), 14–17.

[33] Haixin Wang, Cheng Xu, Xiaojie Chen, Ce Zhang, Haibo Hu, Shikun Tian, Ying Yan, and Jianliang Xu. 2024. V2FS: A Verifiable Virtual Filesystem for Multi-Chain Query Authentication. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 1999–2011.

[34] Haixin Wang, Cheng Xu, Ce Zhang, Jianliang Xu, Zhe Peng, and Jian Pei. 2022. vChain+: Optimizing verifiable blockchain boolean range queries. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1927–1940.

[35] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. 2011. DBA: A dynamic Bloom filter array for scalable membership representation of variable large data sets. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 466–468.

[36] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[37] Zhiqiang Wu and Kenli Li. 2019. VBTree: forward secure conjunctive queries over encrypted data for cloud computing. *The VLDB journal* 28, 1 (2019), 25–46.

[38] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*. 141–158.

[39] YCharts. 2025. Bitcoin Blockchain Size. https://ycharts.com/indicators/bitcoin_blockchain_size.

[40] YCharts. 2025. Ethereum Blockchain Size. https://ycharts.com/indicators/ethereum_chain_full_sync_data_size.

[41] Meihui Zhang, Zhongle Xie, Cong Yue, and Ziyue Zhong. 2020. Spitz: a verifiable database system. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3449–3460. https://doi.org/10.14778/3415478.3415567

[42] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2017. An expressive (zero-knowledge) set accumulator. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 158–173.

[43] Yanchao Zhu, Zhao Zhang, Cheqing Jin, and Aoying Zhou. 2020. Enabling generic verifiable aggregate query on blockchain systems. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 456–465.

[44] Yanchao Zhu, Zhao Zhang, Cheqing Jin, Aoying Zhou, and Ying Yan. 2019. SEBDB: semantics empowered blockchain database. In *2019 IEEE 35th international conference on data engineering (ICDE)*. IEEE, 1820–1831.

A PROOF

A.1 Proof of Theorem 5.3

Theorem 5.3. *Astray tree frequency f_n is a Catalan number C_n.*

Proof. Given an astray tree with n internal nodes, it consists of three parts: the left child branch, the root node, and the right child branch. Then, a normal node in VO can be viewed as a special case of an astray tree where both its child branches are empty, and the root node itself is a leaf, *i.e.*, with $n = 0$ internal node. Otherwise, the frequency f_n for an astray tree with $n > 0$ can be determined by the combination of its child branches recursively as below

$$\begin{cases} f_0 = 1 \\ f_{n+1} = \sum_{i=0}^{n} f_i \cdot f_{n-i} \end{cases} \tag{3}$$

Thus, frequency numbers $\{f_n\}$ satisfy the recurrence relation of Catalan numbers. It can be computed as

$$f_n = C_n = \frac{(2n)!}{n!(n+1)!} \tag{4}$$

□

A.2 Proof of Theorem 5.5

Theorem 5.5. *The sum of relative size is* $r_n = \frac{C_n + C_{n+1}}{2}$.

Proof. Given an astray tree with n internal nodes, it consists of three parts: the root node, the left and right child branches. For $n = 0$, it has an empty child, thus we have $r_0 = 1$. For $n > 0$, the sum of relative size r_n can be computed by the combination of its child branches and root hints recursively as below.

$$r_{n+1} = \frac{1}{2} f_{n+1} + \frac{1}{2} \sum_{i=0}^{n} (r_i f_{n-i} + r_{n-i} f_i) = \frac{1}{2} f_{n+1} + \sum_{i=0}^{n} r_i f_{n-i} \quad (5)$$

In Eq. (3), we have $f_{n+1} = \sum_{i=0}^{n} f_i \cdot f_{n-i}$ and $f_0 = 1$, thus we have

$$f_{n+2} = f_0 f_{n+1} + \sum_{i=0}^{n} f_{i+1} f_{n-i} = \sum_{i=0}^{n} f_i f_{n-i} + \sum_{i=0}^{n} f_{i+1} f_{n-i}$$
$$= \sum_{i=0}^{n} (f_i + f_{i+1}) f_{n-i} \quad (6)$$

Consequently, we have $\frac{f_{n+1}+f_{n+2}}{2} = \frac{1}{2} f_{n+1} + \sum_{i=0}^{n} \frac{f_i + f_{i+1}}{2} f_{n-i}$, which has the same recursive pattern as Eq. (5), *i.e.*, $r_{n+1} = \frac{f_{n+1}+f_{n+2}}{2}$. Since $\{f_n\}$ are Catalan numbers and $r_0 = \frac{f_0+f_1}{2} = 1$, we can conclude that

$$r_n = \frac{C_n + C_{n+1}}{2} \quad (7)$$

□