Data Science and Engineering (2026) 11:177–198
https://doi.org/10.1007/s41019-025-00326-6

RESEARCH PAPERS

BlockSketch: A Hybrid Tree-Based Sketch for Keyword Search in Blockchain Systems

Yushi Liu[1] · Xiaodong Qi[2] · Zhao Zhang[1] · Yanqin Yang[1] · Cheqing Jin[1] · Aoying Zhou[1]

Received: 11 July 2025 / Revised: 10 October 2025 / Accepted: 2 November 2025 / Published online: 13 January 2026

Abstract
Keyword search, which identifies transactions associated with specified keywords across historical blocks, is a critical query type in blockchain analytics. However, existing approaches, such as on-chain indexing and off-chain synchronization, may lead to significant space overhead or challenges in maintaining data freshness. To address these challenges, we propose BlockSketch, a novel probabilistic data structure (PDS) that adopts a differentiated encoding strategy, aimed at resolving the trade-off between query performance and storage overhead in blockchain indexing. BlockSketch features a hierarchical filtering architecture that combines Bloom filters and Sketches within a binary tree framework, enabling dynamic structural maintenance. Keywords are categorized as "hot" or "cold" based on their on-chain frequency and encoded into the most suitable component to achieve resource-efficient storage and accurate querying. In addition, BlockSketch integrates two distinct query rules, namely "level-down" and "jump," to balance query accuracy and efficiency when processing keywords with varying frequencies. Furthermore, we enhance the query efficiency of BlockSketch by merging inefficient lower-level nodes into more compact ones and pruning redundant node checks during query execution. Extensive experiments on a real-world dataset demonstrate that BlockSketch delivers up to 73% faster query processing, achieves 44.56% of the average false positive rate of baselines at low multiplicity and as low as 1.52% at high multiplicity, and saves 79% in storage compared to state-of-the-art methods.

Keywords Blockchain · Keyword search · Bloom filter · Sketch

1 Introduction

The rapid advancement of blockchain technologies [1, 2] has enabled a wide range of applications[3–5]. The exponential growth in transaction volumes poses significant regulatory challenges [6] and necessitates efficient mechanisms for monitoring and analysis [7]. Keyword search, as a prevalent technique enabling this capability, has been extensively studied in database systems [8–10]. Its applicability to blockchain supervision stems from the fact that on-chain transactions can be modelled as a keyword set encompassing both transaction-native keywords (e.g., sender, receiver) and contract-specific keywords. Consequently, the audit of anomalous transactions can be transformed into the detection of anomalous keywords within transactions. For example, as shown in Fig. 1, a DeFi analyst, who is tracking a phishing account David that sends messages to random accounts, may use keyword search to locate those transactions containing the keyword "David". However, existing blockchain systems are inefficient for such tasks. Their data

✉ Yanqin Yang
yqyang@dase.ecnu.edu.cn

Yushi Liu
liuys@stu.ecnu.edu.cn

Xiaodong Qi
xiaodong.qi@ntu.edu.sg

Zhao Zhang
zhzhang@dase.ecnu.edu.cn

Cheqing Jin
cqjin@dase.ecnu.edu.cn

Aoying Zhou
ayzhou@dase.ecnu.edu.cn

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China

[2] School of Computer Science and Engineering, Nanyang Technological University, Singapore, Singapore

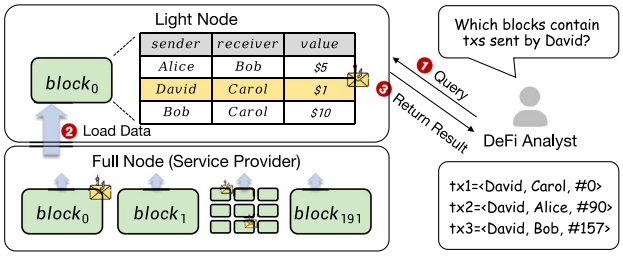


Fig. 1 An example of phishing account tracking

retrieval interfaces are simplistic due to the lack of a built-in secondary index for content. Consequently, any keyword search necessitates a full sequential scan of the specified block range, leading to prohibitive costs from processing irrelevant data. Meanwhile, since it is impossible to predict which blocks contain relevant transactions, all the blocks during the period must be checked, resulting in unnecessary cost on processing irrelevant "normal" blocks.

Although several studies focus on on-chain index structures [11–15] — such as B+ trees[16], inverted indexes [17], and other indexing schemes — to accelerate queries upon blockchain systems, these approaches typically suffer from high space overhead and sophisticated maintenance cost. Similarly, approaches that synchronize on-chain data with off-chain databases for analytics [18–21] double storage overhead and require efficient synchronization to remain up-to-date, which makes the system more complex. Locating transactions associated with a specific keyword requires verifying the presence of the keyword across multiple blocks, which is conceptually similar to the well-studied problem of *Multi-Set Multi-Membership* query—that is, determining which sets contain the given element[22]. Probabilistic data structures (PDSs), such as the Bloom filter[23], Cuckoo filter[24], and others[25–27], can address this challenge by minimizing storage overhead while maintaining relatively high query accuracy.

However, existing PDSs face several limitations in blockchain environments. First, they typically rely on preallocated disk space, inefficient for blockchain data that arrives continuously block by block. Preallocating a large space can lead to significant waste, while insufficient space may necessitate costly reconstructions when capacity is exceeded. For instance, a standard Bloom filter must be pre-allocated with a fixed-size bit array. As more elements are encoded, the false positive rate degrades, necessitating a costly resizing operation[28]. This process typically involves doubling the allocated space, which results in substantial space wastage, especially when the original capacity is large. Second, the accuracy of PDSs heavily depends on the multiplicity of the queried element, i.e., the number of sets it belongs to. The keywords of blockchain are unevenly distributed. For example, in RAMBO[25], each filter group maps an element to a separate Bloom filter based on the set ID, and the qualifying sets are then identified by intersecting the query results from these multiple groups. However, if an element has a high multiplicity, it will cause nearly every individual Bloom filter to return "true". Consequently, the intersection operation becomes almost ineffective at filtering, leading to an extremely high false positive rate in the final query result. In the context of blockchain systems, a high false positive rate is particularly detrimental. Each false positive compels the system to perform a costly operation: retrieving a full block from disk, followed by deserialization and a full scan of its transactions, only to find that the target keyword is not present. Therefore, achieving high accuracy is not merely a desirable trait but a critical objective for ensuring query performance.

To overcome these obstacles, we propose *BlockSketch*, a novel probabilistic data structure supporting keyword search in blockchain systems with high accuracy and low storage cost. Instead of employing a large PDS encoding all keywords within the block range, or assigning a separate filter to each block, BlockSketch adopts a hierarchical search tree, which uses a jump-like encoding schema to encode keywords along the path from a leaf node to the root. Each node integrates a Bloom filter and a Sketch, a more advanced filtering structure, to differentiate hot and cold keywords within a block. Specifically, the Bloom filter encodes hot keywords that occur frequently during maintenance, while the Sketch handles membership queries for keywords with relatively low multiplicity. By distributing keywords across multiple nodes, the number of keywords encoded by each node's Bloom filter and Sketch is limited, resulting in smaller data structures and improved efficiency. Furthermore, the reduced multiplicity of each keyword enhances query accuracy. Leveraging its hierarchical structure, BlockSketch enables rapid identification of all potential results. BlockSketch adopts a bottom-up dynamic construction scheme to ensure efficient maintenance and high space utilization, and indexes blocks at a window-level granularity to enhance query efficiency. Additionally, BlockSketch merges lower-level nodes into a single flat node and integrates a tree-based pruning strategy, significantly reducing query latency while maintaining space efficiency.

To summarize, our contributions are as follows:

- We present BlockSketch, a novel probabilistic data structure tailored for efficient keyword search in blockchain systems. BlockSketch incorporates a dynamic structure maintenance strategy to accommodate the continuously growing block data. Its differentiated encoding strategy effectively reduces the storage overhead of hot keywords compared with the level-by-level coding

schema, while also improving query efficiency by integrating both "level-down" and "jump" query strategies.
- We enhance the query efficiency of BlockSketch with a twofold optimization. On one hand, we introduce "flat nodes" to flatten the lower-level subtrees into compact nodes, reducing the time spent at these levels. On the other hand, we design a tree-based pruning strategy that identifies and eliminates branches in the search path that are unlikely to contain true-positive results, further reducing the search space.
- We implement a prototype of BlockSketch, and conduct extensive experiments on real-world datasets against state-of-the-art methods to evaluate the effectiveness and efficiency of the proposed structure.

The rest of the paper is organized as follows. Section 2 reviews the related works relevant to our research. Section 3 introduces the problem definition and provides a system overview. Section 4 details the architecture of BlockSketch and the maintenance mechanism. In Sect. 5, we describe the query processing workflow and present a twofold optimization strategy. Section 6 covers the implementation of BlockSketch, reports the experimental results, and Sect. 7 concludes the paper.

2 Related Work

In this section, we summarize the relevant research work on blockchain data query and probabilistic data structures in recent years.

2.1 Blockchain Data Query

Blockchain technology is a decentralized and immutable ledger system that ensures secure and transparent data management through a distributed network. The system continuously extends the ledger by appending a new block, a group of transactions agreed upon by the network through a consensus protocol [1, 2, 29]. As shown in Fig. 2, each block consists of a header and a body. The header includes metadata, and the body involves a collection of transactions organized as a Merkle tree [30]. Each transaction contains the sender address, receiver address, timestamp, and other related data. This structure ensures data integrity, prevents tampering, and allows efficient verification, making blockchain a robust solution for secure data management.

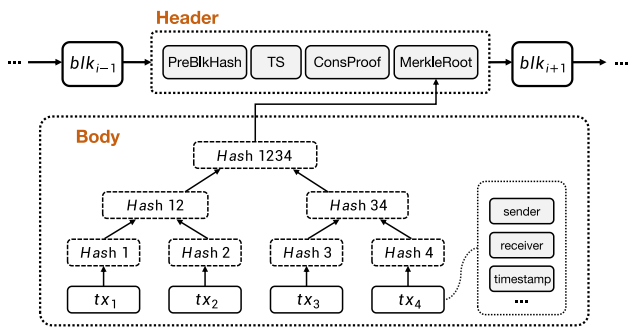


Fig. 2 Blockchain structure

Blockchain systems typically offer only basic query capabilities, prompting research efforts to enhance on-chain query functionality by introducing advanced features. EtherQL [11] adds an efficient query layer to Ethereum, enabling flexible analytical queries such as range query and top-k query. SEBDB [12] integrates relational semantics into blockchain, supporting SQL-like operations to manage both on-chain and off-chain data. In scenarios where blockchain nodes are considered untrustworthy, some studies emphasize verifiable on-chain queries. vChain [13, 14] introduces a verifiable query framework that supports efficient boolean range queries while guaranteeing result integrity. GEM^2-Tree [15] reduces gas consumption during range queries. AMatching [31] ensures soundness and completeness by efficiently verifying subgraph query results in hybrid-storage blockchains. Similarly, MP-DAG [31] efficiently handles authenticated keyword searches by aggregating unqualified paths absent in result trees. Additionally, significant efforts have been made to synchronize on-chain data with off-chain storage engines to support more complex queries. The Graph[1], a decentralized protocol for indexing and querying blockchain data, utilizes PostgreSQL[2] to cache block data, enabling high-performance queries for advanced blockchain applications. BlockchainDB [32] enhances blockchains by leveraging them as a tamper-proof storage layer and introducing a database layer with classical data management techniques, such as sharding. BlockSci [21], an open-source blockchain analysis platform, features an in-memory database optimized for sequential and graph traversal queries, delivering superior performance compared to general-purpose graph databases. However, these approaches require substantial additional storage and rely on maintaining up-to-date off-chain data.

2.2 Probabilistic Data Structure and Membership Query

A *Bloom filter* is a probabilistic, space-efficient data structure used to determine whether an element is a member of a set [23], trading space efficiency for a small probability of false positives, where it may incorrectly indicate that an element is in the set. Existing work enhances the functionality

[1] https://thegraph.com.

[2] https://www.postgresql.org.

of Bloom filters by introducing auxiliary information or integrating them with other data structures. The Counting Bloom Filter [33] addresses the limitation of not supporting element deletion by adding a counter for each bit. The Hierarchical Bloom Filter [34] extends Bloom filters by splitting input strings into fixed-size blocks, allowing efficient substring matching. The Stable Bloom Filter [35], designed for unbounded data streams, removes stale information to manage limited space while maintaining a bounded false positive rate, effectively handling duplicates. Additionally, other variants have been developed to support scalability [28, 36], association and multiplicity queries [37], and range queries [38].

Unlike the single-set membership query supported by Bloom filters, the Multi-Set Multi-Membership query (MS-MMq) aims to determine whether an element belongs to any set within a collection of multiple sets, which is the focus of this work. BUFFALO [39] and BIGSI [27] assign each set a Bloom filter, requiring queries to access multiple Bloom filters, leading to increased space usage and slower query speed. BloomTree [26] introduces a tree structure based on Bloom filters optimized for efficient multiple-set membership queries. The Combinatorial Bloom Filter [40] minimizes space usage without prior knowledge of set sizes and supports efficient hardware implementations using multiple hash functions and combinatorial coding. The Noisy Bloom Filter [41] introduces noise to reduce storage consumption, and its extension, NBF-E, employs error-correcting codes to mitigate the effects of noise, thereby improving MS-MMq efficiency. RAMBO [25] combines Bloom filters with the Count-Min Sketch technique to enhance accuracy and query speed, while CSC [22] encodes element presence and set information into a compact sketch structure that integrates with mainstream data structures like Bloom filters and Cuckoo filters. However, directly applying these approaches to an entire window would result in high false positive rates and significant storage overhead. Instead, they can be effectively integrated as the Sketch component within BlockSketch to fully leverage their strengths while mitigating these limitations. In this paper, we refer to the structures based on the *Count-Min Sketch* [42] as "sketches", which serve as the fundamental building block of our proposed approach.

3 Problem Definition and Approach Overview

In this section, we provide the definition of the problem and an overview of the proposed approach.

3.1 Problem Definition

With the increasing adoption of blockchain technology, keyword search, which retrieves transactions related to a keyword within a certain block range, becomes quite common. This problem in blockchain systems can be formalized as a membership query across multiple sets with a temporal constraint. To simplify, a transaction *tx* is expressed as:

$$tx = \langle K, bid, ts, payload \rangle,$$

where K represents the set of keywords $\{\kappa_1, \kappa_2, ...\kappa_n\}$, *bid* denotes the block identifier to which *tx* belongs, *ts* is the transaction timestamp, and *payload* encapsulates the remaining attributes.

Definition 1 (Keyword Search Query) Given a collection of historical blocks $\mathcal{B} = \{B_1, B_2, ..., B_n\}$, each containing a set of transactions, the keyword search query is to identify all blocks that contain at least one transaction associated with a specific keyword κ within a given time interval $[s, t]$. Formally, the result of a keyword search query $M_\kappa^{[s,t]}$ is defined as:

$$M_\kappa^{[s,t]} = \{B \in \mathcal{B} \mid \exists tx \in B,\ \kappa \in tx.K \wedge tx.ts \in [s,t]\},$$

where κ is the queried keyword, and $[s, t]$ specifies the time range between the start time s and the end time t. In particular, a query without temporal constraints is expressed as $M_\kappa^{[0,\infty)}$.

After identifying the blocks in $M_\kappa^{[s,t]}$, the relevant transactions can be further extracted by examining the transactions within these blocks. This formulation effectively captures the essence of blockchain keyword search, which filters blockchain data under both temporal and keyword-based constraints.

However, designing an efficient structure that supports keyword search is not trivial. A straightforward solution is to utilize a binary search tree, which lets each leaf node correspond to a single block; every internal node has two children. Then, each internal node is equipped with a Bloom filter to indicate whether the blocks this node covers contain transactions with a specified keyword. During a query, the Bloom filter at each node is examined to determine if the target keyword exists in the sub-tree. If the filter indicates a match, the search continues to its children; otherwise, the search terminates at that node. However, as the level rises, a node covers more blocks, and more keywords are encoded into its Bloom filter, leading to two downsides as mentioned in Section 1. (1) The false positive rate increases, resulting in more extra blocks that must be checked to obtain accurate

transactions. (2) A larger Bloom filter is required to accommodate the increasing number of keywords, and each keyword has to be redundantly encoded in the filters from the root to the corresponding leaf node.

In this paper, we propose a novel probabilistic data structure that simultaneously ensures high query accuracy and low space overhead to support keyword search. Specifically, given a keyword search $M_w^{[s,t]}$, the proposed structure aims to identify all true-positive blocks, which are represented by a set $\mathcal{B}_T$. Let $\widehat{\mathcal{B}_T}$ denote the set of candidate blocks (i.e., the set containing both true positives and false positives) returned by the structure. Our objective is to minimize the number of false-positive blocks while ensuring query completeness, which can be formally defined as:

$$\min(|\widehat{\mathcal{B}_T}| - |\mathcal{B}_T|) \quad \text{s.t. } \mathcal{B}_T \subseteq \widehat{\mathcal{B}_T}.$$

In addition, the storage cost of the proposed structure should remain sufficiently low to support scalable and practical deployment.

3.2 Overview

Figure 3 presents an overview of the proposed method, which involves two core layers in blockchain systems: (1) the indexing layer, responsible for constructing BlockSketch for specific keywords in the blockchain; and (2) the query layer, which retrieves relevant index structures and filters results based on query constraints. First, BlockSketch is designed for the append-only nature of blockchains, where it must continuously incorporate new data. To balance query performance against the escalating storage demands of a growing chain, we persist the BlockSketch on disk rather than keeping the entire structure in memory. Each new block triggers an append-only write operation. To further enhance efficiency and scalability, we partition the blockchain into consecutive windows. Each window is indexed by its dedicated BlockSketch. When a window is full, its BlockSketch becomes immutable, and a new one is created for subsequent blocks. During a query, we first identify the keyword and block range constraints, then we locate the corresponding windows covering the range. The query may span multiple windows, as shown in the example in Figure 1, which covers the block range [0, 191]. In such cases, the system loads the BlockSketch for each relevant window, executes the sub-queries concurrently, and merges the results. Each sub-query returns a set of candidate blocks, which may contain false positives (highlighted in red). A final verification step on these candidate blocks retrieves the precise target transactions. In summary, the window-based indexing strategy brings dual benefits: (1) Instead of incurring heavy I/O overhead from loading memory-exceeding monolithic structures, only smaller necessary units are accessed, significantly reducing disk operations and avoiding memory swapping; (2) Non-overlapping data across windows enables full utilization of parallelization techniques to enhance query efficiency, ensuring system scalability and responsiveness.

4 Construction of BlockSketch

To address the aforementioned challenges, we propose a novel probabilistic data structure, BlockSketch, to support efficient keyword search on blockchain data.

4.1 Fundamental Design of BlockSketch

The structure of BlockSketch is shown in the right part of Figure 3. BlockSketch adopts a selective encoding strategy, where keywords are only encoded at specific levels, avoiding their encoding all keywords at every level. This strategy can effectively reduce the Bloom filter's storage cost, which depends on the number of keywords encoded, and enhance query accuracy. Furthermore, each node in BlockSketch is associated with a Sketch that records the next node where the target keyword is encoded. The jump-like traversal significantly reduces the number of nodes accessed, thereby improving query efficiency. The hybrid design achieves a substantial reduction in storage overhead while maintaining high query accuracy and search performance. Specifically, a BlockSketch structure has two types of nodes: (1) leaf node, and (2) index node. A leaf node logically represents a single block and has no other fields except the block identifier. An index node represents a larger block range that covers the blocks of its descendant leaf nodes. It also contains the

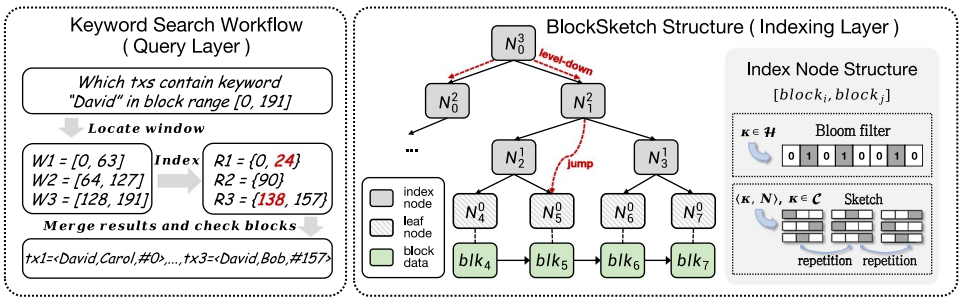


Fig. 3 The query workflow and structure of BlockSketch

pointers of its child nodes, a Bloom filter $\mathcal{F}$, and a Sketch $\mathcal{S}$. Nodes are constructed in a bottom-up manner: leaf nodes at level *i* are paired to form the parent nodes at level $i+1$, and this process continues iteratively until the root node. During the construction process, determining the appropriate level at which a keyword should be encoded in the Bloom filter and establishing jump relationships between nodes using Sketches are essential. To achieve this, we first introduce some fundamental concepts as follows:

Definition 2 (Keyword Set) Given a leaf node N_{leaf}, the keyword set $\mathrm{Set}(N_{leaf})$ is the set of keywords involved in the transactions referenced by N_{leaf}. Moreover, for an index node N_p with two child nodes, N_l and N_r, the keyword set is defined as $\mathrm{Set}(N_p) = \mathrm{Set}(N_l) \cup \mathrm{Set}(N_r)$.

Definition 3 Given an index node N_p with two child nodes, N_l and N_r, a keyword κ is considered locally hot if it exists in both $\mathrm{Set}(N_l)$ and $\mathrm{Set}(N_r)$. Otherwise, it is classified as a locally cold keyword. The sets of locally hot keywords, $\mathcal{H}$, and locally cold keywords, $\mathcal{C}$, are formally defined as follows:

$$\mathcal{H}(N_p) = \mathrm{Set}(N_l) \cap \mathrm{Set}(N_r),\ \mathcal{C}(N_p) = \mathrm{Set}(N_l) \cup \mathrm{Set}(N_r) - \mathcal{H}(N_p).$$

We define a keyword κ as "locally hot" only when it appears in both child nodes, a design choice primarily motivated by optimizing space efficiency in the subsequent encoding process. If, alternatively, we were to define a keyword present in just one child node as hot, this "hot" status would have to be propagated and stored at every node along the path up to the root. In contrast, our definition ensures that the "locally hot" status is recorded only at the confluence where both children contain κ, and the "locally cold" status propagates upwards and will be finally recorded until a transition from cold to hot occurs. This approach leads to substantial space savings during the structure maintenance phase, as will be detailed in Sect. 4.2.

Definition 4 (Jump Node) Given a node *N* at level *u* and a keyword κ, the jump node $\mathcal{J}(\kappa, N)$ is defined as the highest-level descendant node N_d of *N* that is located at level *u* or below, such that $\kappa \in \mathcal{H}(N_d)$ if N_d is an index node, or $\kappa \in \mathrm{Set}(N_d)$ if N_d is a leaf node.

If a keyword κ is locally hot with respect to a node *N*, both branches of *N* must be traversed to locate κ, causing the search path to fork and requiring exploration of both child nodes. Conversely, if κ is locally cold, the search follows a single path to a specific node, i.e., the jump node defined above, where it either diverges again or terminates. To optimize the search process, κ is encoded in the Bloom filter of *N* only when it is locally hot. Otherwise, the Sketch of *N* records the jump node $\mathcal{J}(\kappa, N)$, enabling the search to bypass intermediate nodes and minimize unnecessary checks.

Based on the above design choice, the inspection of BlockSketch exhibits different patterns depending on whether the queried keyword is "cold" or "hot," to maintain both query efficiency and accuracy. Specifically, when the keyword is cold, the query focuses mainly on the Sketch component of BlockSketch. With the help of the jump node, the candidate blocks can be quickly restricted to a much smaller range, allowing many intermediate nodes to be skipped and thereby reducing the search space. In addition, since the multiplicity is low, which is consistent with the characteristic of Sketches in handling low-multiplicity elements with very low false positives, the accuracy of BlockSketch can be guaranteed. When the keyword is hot, more Bloom filters may return positive results during the query process, which directly leads to more nodes being inspected compared with the cold-keyword case. Nevertheless, due to the binary tree structure of BlockSketch, the query complexity still remains at the order of $O(\log n)$. Meanwhile, although Bloom filters inherently produce false positives, a leaf node will only be added to the result set if every node along the entire path from the first false positive down to that leaf also produces false positives. As a result, the impact of a single false positive is progressively diminished as the search descends, making the final query results more reliable.

4.2 Dynamic Structure Maintenance

As we discussed previously, a key limitation of existing methods is that they require a priori knowledge of the dataset's size. Consequently, they cannot dynamically construct the index as new data arrives in real-time, which makes it difficult to achieve an optimal configuration. The fundamental reason for this deficiency is that these methods are "holistic" in nature; they encode the membership relations of all elements across all sets into their internal filter units via hash functions. Any attempt to modify the structure would affect the already encoded elements, making a full reconstruction the only viable update strategy. To handle the continuously arriving data in a blockchain, we must design an "incremental" structure that can maintain an optimal parameter configuration throughout the construction process. A tree-based structure is precisely such a design that enables incremental updates. The arrival of a new block is treated as the addition of a new leaf node at the rightmost edge of the tree. This operation only affects a single path from the new leaf up to the root. The vast majority of existing nodes and subtrees that are not on this update path remain entirely unaffected, meaning their internal filters do not need to be rebuilt

in response to new data. Based on this insight, we design a dynamic structure maintenance algorithm that enables BlockSketch to ingest new block data without altering the already constructed parts of the index.

When a new block arrives, BlockSketch creates a new leaf node N_r to represent it. If N_r's left sibling node N_l does not yet have a parent node (i.e., it is unpaired), a new internal node N_p is created at the next higher level through the following steps. First, the parent-child relationship is established by linking N_p to both N_l and N_r. Second, the locally hot keyword set $\mathcal{H}(N_p)$, shared between the keyword sets of N_l and N_r, is identified and inserted into the Bloom filter associated with N_p. By encoding only these locally hot keywords in the Bloom filter, its size is significantly reduced, thereby enhancing space efficiency. Following this process, BlockSketch continues constructing parent nodes at higher levels in the same manner.

Additionally, for a parent node N_p at a level greater than 2 with child nodes N_l and N_r, BlockSketch performs an additional operation: for each keyword $\kappa \in \mathcal{H}(N_p) \wedge \kappa \notin \mathcal{H}(N_l)$, a tuple $\langle \kappa, \mathcal{J}(\kappa, N_l) \rangle$ is inserted into the Sketch of N_l, and a similar operation is applied to N_r. This operation is crucial because when querying a keyword κ at node N_p, κ is locally hot in N_p, requiring the search to proceed to both N_l and N_r. If κ is locally cold in either N_l or N_r, the Sketch is used to identify the jump node $\mathcal{J}(\kappa, N_l)$ or $\mathcal{J}(\kappa, N_r)$, allowing the search to bypass intermediate nodes and directly jump to the target node, thereby improving query efficiency. Note that the tuple $\langle \kappa, N \rangle$ with keyword κ is only encoded once by the Sketch of each node. As mentioned in Section 2, if the multiplicity of the keyword is low, the accuracy increases accordingly, leading to a low false positive rate.

Calculating the jump node for each keyword from scratch at each node is computationally intensive and complex. To optimize this process and reduce unnecessary computations, BlockSketch attaches a temporary buffer to each node to maintain the jump nodes for all keywords, updating it incrementally during construction. For a leaf node N_{leaf}, the buffer $\mathrm{Buf}(N_{leaf})$ stores a tuple $\langle \kappa, N_{leaf} \rangle$ for each keyword $\kappa \in \mathrm{Set}(N_{leaf})$, where N_{leaf} itself is the jump node for all corresponding keys. For an index node N_p with two child nodes N_l and N_r, the jump node for a keyword $\kappa \in \mathrm{Set}(N_p)$ is determined as follows. If $\kappa \notin \mathcal{H}(N_p)$, meaning it exists only in one child' keyword set (e.g., N_l), then $\mathcal{J}(\kappa, N_p) = \mathcal{J}(\kappa, N_l)$, and the tuple $\langle \kappa, \mathcal{J}(\kappa, N_l) \rangle$ is added to $\mathrm{Buf}(N_p)$. If $\kappa \in \mathcal{H}(N_p)$, indicating it is locally hot and the search path for it diverges at N_p, the jump node $\mathcal{J}(\kappa, N_p)$ is N_p itself, and the tuple $\langle \kappa, N_p \rangle$ is added to $\mathrm{Buf}(N_p)$. Notably, these temporary buffers are released once they are no longer needed.

BlockSketch implements a dynamic space allocation mechanism by initializing the Bloom filter and Sketch in each index node only after the keywords to be encoded have been identified. This strategy allows their sizes to be adjusted dynamically based on the number of keywords, ensuring efficient space utilization without the need to pre-allocate fixed space.

Example 1 Figure 4 illustrates the construction process of BlockSketch. Initially, the structure contains three leaf nodes, for example, N_1^0, where the superscript 0 denotes the node's level and the subscript 1 is the identifier of its corresponding block. For each key κ in the keyword set of a leaf node N_{leaf}, a pair $\langle \kappa, N_{leaf} \rangle$ is constructed and stored in a temporary buffer associated with the node. For instance, the buffer of N_1^0 is $\mathrm{Buf}(N_1^0) = \{\langle \mathrm{Bob}, N_1^0 \rangle\}$. The key operations on Bloom filters and sketches proceed as follows: (1) Before N_3^0 is added, only the hot keyword Bob (for N_0^0 and N_1^0) is encoded into the Bloom filter of the new parent node N_0^1. This creates a new pair $\langle \mathrm{Bob}, N_0^1 \rangle$, which is merged into $\mathrm{Buf}(N_0^1)$. Meanwhile, the pair $\langle \mathrm{Alice}, N_0^0 \rangle$ is added unmodified to $\mathrm{Buf}(N_0^1)$. Subsequently, $\mathrm{Buf}(N_0^0)$ and $\mathrm{Buf}(N_1^0)$ are released. As N_2^0 has no sibling, it becomes a root and its buffer persists. (2) Upon adding N_3^0, $\mathrm{Buf}(N_2^0)$ and $\mathrm{Buf}(N_3^0)$ are first merged into a new buffer for their parent N_1^1. Next, the Bloom filter of the new higher-level node N_0^2 encodes the key Alice, generating the pair $\langle \mathrm{Alice}, N_0^2 \rangle$. $\mathrm{Buf}(N_0^2)$ will be retained until its parent node's construction finishes. (3) Since Alice is hot in both N_0^1 and N_1^1,

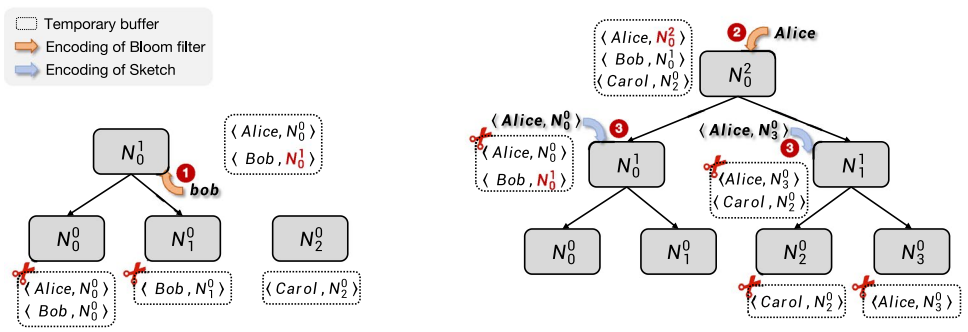


Fig. 4 An example of structure construction

its corresponding pairs ($\langle \text{Alice}, N_0^0 \rangle$ and $\langle \text{Alice}, N_3^0 \rangle$) are encoded into the respective sketches of N_0^1 and N_1^1. At this stage, N_0^2 remains the sole root node. To minimize memory overhead, all buffers except $\text{Buf}(N_0^2)$ can be released.

Once the number of leaf nodes in BlockSketch reaches its capacity, no new blocks are added. At this stage, all keywords from different blocks converge at the root node and are encoded into the Sketches and Bloom filters. The BlockSketch is then archived by clearing the root's buffer. Notably, during construction, BlockSketch may not form a single-tree structure if the number of leaf nodes is not a power of two. For example, with three blocks, three leaf nodes N_0^0, N_1^0, and N_2^0 are created at level 0, while only one parent node N_0^1 is formed at level 1 for N_0^0 and N_1^0, resulting in N_0^1 and N_2^0 serving as the roots of two separate sub-trees. Query processing for such intermediate and incomplete BlockSketch structures is discussed later.

Algorithm 1 outlines the bottom-up construction process of BlockSketch within each window. The process relies on two global data structures, M and $\mathcal{R}$, where $\mathcal{R}$ stores the roots of sub-trees during construction. Initially, $\mathcal{R}$ is empty (line 1). The main construction logic is executed through lines2 to 31, triggered whenever a new block is added. For each incoming block B_i, BlockSketch creates a new leaf node N_{leaf} corresponding to B_i, and initializes its buffer $\text{Buf}(N_{leaf})$ by setting the jump node of each keyword in $\text{Set}(N_{leaf})$ to N_{leaf} itself (line 3). Subsequently, BlockSketch iteratively constructs parent nodes for each pair of sibling nodes in a level-by-level manner (lines 4 to 22)..

First, if the node N_r does not have an unpaired left sibling node, N_r is identified as the root of the current subtree, and BlockSketch terminates the procedure (line 5 to 8). This is because a node can only pair with its left sibling. Otherwise, BlockSketch retrieves the left sibling N_l and creates their parent node N_p (line 9). Next, BlockSketch

Input: M: maximum number of leaf nodes; B_i: newly arrived block to be indexed
Output: $\mathcal{R}$: root nodes of BlockSketch

1: $\mathcal{R} \leftarrow \emptyset$
2: **procedure** onBlockArrival(B_i)
3: $N_{leaf} \leftarrow$ `create_leaf`(B_i) , $N_r \leftarrow N_{leaf}$, $u \leftarrow 0$
4: **while true do**
5: **if** N_r has no unpaired left sibling node N_l at level u **then**
6: $\mathcal{R} \leftarrow \mathcal{R} \cup \{N_r\}$
7: **break**
8: **end if**
9: $N_p \leftarrow$ `create_node`(N_l, N_r)
10: $\mathcal{H}(N_p) \leftarrow$ `Set`(N_r) $\cap$ `Set`(N_l)
11: $\mathcal{C}(N_p) \leftarrow$ `Set`(N_r)Δ`Set`(N_l)
12: `insert_bloom_filter`($N_p.\mathcal{F}, \mathcal{H}(N_p)$)
13: **if** neither N_l nor N_r is a leaf node **then**
14: $S_l \leftarrow \{\langle \kappa, \mathcal{J}(\kappa, N_l) \rangle \mid \kappa \in \mathcal{H}(N_p) \wedge \kappa \notin \mathcal{H}(N_l)\}$
15: `insert_sketch`($N_l.\mathcal{S}, S_l$)
16: $S_r \leftarrow \{\langle \kappa, \mathcal{J}(\kappa, N_r) \rangle \mid \kappa \in \mathcal{H}(N_p) \wedge \kappa \notin \mathcal{H}(N_r)\}$
17: `insert_sketch`($N_r.\mathcal{S}, S_r$)
18: **end if**
19: Merge_Buffer(N_p, N_l, N_r)
20: $N_r \leftarrow N_p$, $u \leftarrow u + 1$
21: $\mathcal{R} \leftarrow \mathcal{R} \setminus \{N_l\}$
22: **end while**
23: $\mathcal{R} \leftarrow \mathcal{R} \cup \{N_r\}$
24: **if** the number of leaf nodes reaches capacity M **then**
25: $N_{root} \leftarrow$ `get_root`($\mathcal{R}$)
26: $S \leftarrow \{\langle \kappa, \mathcal{J}(\kappa, N_{root}) \rangle \mid \kappa \in \mathcal{C}(N_{root})\}$
27: `insert_sketch`($N_{root}.\mathcal{S}, S$)
28: `delete_buf`(N_{root})
29: **end if**
30: **return** $\mathcal{R}$
31: **end procedure**
32:
33: **function** Merge_Buffer(N_p, N_l, N_r)
34: $buf \leftarrow \{\langle \kappa, \mathcal{J}(\kappa, N) \rangle \in$ `Buf`(N_l) $\cup$ `Buf`(N_r) $\mid \kappa \in \mathcal{C}(N_p)\}$
35: $buf \leftarrow buf \cup \{\langle \kappa, N_p \rangle \mid \kappa \in \mathcal{H}(N_p)\}$
36: `delete_buf`(N_l), `delete_buf`(N_r)
37: **end function**

Algorithm 1 BlockSketch construction over blocks

computes the hot keyword set $\mathcal{H}(N_p)$ and the cold keyword set $\mathcal{C}(N_p)$ based on $\mathrm{Set}(N_l)$ and $\mathrm{Set}(N_r)$, and initializes a Bloom filter for N_p with $\mathcal{H}(N_p)$ (line 12). The Sketch in N_l or N_r is then used to identify the jump nodes of keywords in $\mathrm{Set}(N_p)$ that are hot in N_p but cold in N_l and N_r (line 13 to 18). Subsequently, the buffers $\mathrm{Buf}(N_l)$ and $\mathrm{Buf}(N_r)$ are merged to create $\mathrm{Buf}(N_p)$, which records the jump nodes of all keywords for N_p, using the $\mathrm{Merge\_Buf}()$ function (line 19). During this process, the jump nodes for cold keywords remain unchanged, while N_p is assigned as the new jump node for hot keywords (line 34 to 35). Once the buffer merge is completed, the buffers of the child nodes N_l and N_r are deleted to conserve memory (line 36).

Afterward, BlockSketch proceeds to construct a new parent node at a higher level and removes the old root N_l of the sub-tree. If all nodes are created, the new root N_r of the sub-tree is added to $\mathcal{R}$ (lines 23). Finally, when BlockSketch reaches its predefined capacity, the construction process for the current window's BlockSketch concludes (line 24 to 29).

4.3 Analysis

We then analyze the storage overhead of BlockSketch. Given a keyword κ with multiplicity v, if κ appears multiple times within the same set, its contribution to the overall multiplicity of κ is 1. Assuming the number of leaf nodes in BlockSketch is M and the height of BlockSketch is $\log_2 M + 1$, κ can only be encoded at levels 0 through $\log_2 M$. We define the space overhead of encoding κ in a Bloom filter as $cost_b(\kappa)$ and in a Sketch as $cost_s(\kappa)$.

When $v = 1$, κ remains cold within any node, and its jump node always corresponds to its designated leaf node. This jump node is propagated through the buffers of various internal nodes until it is ultimately recorded in the root's Sketch. In this case, the space overhead for encoding κ is $cost_s(\kappa)$. When $v > 1$, κ is initially cold in a leaf node N_{leaf}. As the tree is built bottom-up, κ becomes a locally hot keyword in N, which is an ancestor node of N_{leaf}. The locally hot keyword is then encoded in a Bloom filter with an overhead of $cost_b(\kappa)$, and the encoding procedure of the Sketch will be triggered twice, resulting in a combined space overhead of $2cost_s(\kappa)$. Therefore, the total space cost of a transition from a locally cold keyword to a locally hot keyword is:

$$cost_b(\kappa) + 2cost_s(\kappa).$$

This transition occurs $v - 1$ times during the maintenance of κ. If, in the final transition, the node N, whose Bloom filter encodes κ, is not the root node of BlockSketch, the pair $\langle \kappa, N \rangle$ must be propagated along the path from N to the root and ultimately encoded into the root's Sketch. Therefore, the total space overhead required for encoding κ is:

$$(v - 1) \cdot (cost_b(\kappa) + 2cost_s(\kappa)) + \delta \cdot cost_s(\kappa),$$

where $\delta = 1$ if an additional encoding in the Sketch of the root node is needed; otherwise, $\delta = 0$.

In contrast, while the BloomTree scheme exclusively uses space-efficient Bloom filters, for a single keyword κ, a node at every level of the tree must encode it. Therefore, the space overhead for encoding κ depends not only on its multiplicity v but also on the height of the tree. This relationship can be approximated as:

$$v \cdot (\log_2 M + 1) \cdot cost_b(\kappa).$$

If we simplify the space complexity of encoding κ in a single Bloom filter or Sketch to a constant $O(1)$, we can observe that the complexity for BlockSketch is $O(v)$, while for BloomTree it is $O(v \cdot \log M)$. Consequently, when the height of the tree is large, the space savings of BlockSketch become particularly significant. In summary, BlockSketch exhibits two key characteristics regarding space usage: (1) The storage overhead for encoding a keyword depends only on its multiplicity, independent of the tree's height; (2) BlockSketch is well-suited for both low-multiplicity datasets and those with uneven distributions. For low-multiplicity keywords, it maintains a minimal disk footprint, while for high-multiplicity keywords, it allocates additional storage space to achieve high query accuracy.

5 Query Processing

In this section, we first demonstrate how BlockSketch efficiently supports the intra-window query in Section 5.1. Given that a query's interval may span multiple windows, we then outline the workflow for inter-window queries in Section 5.2.

5.1 Intra-Window Query Processing

We utilize the depth-first search (DFS) strategy to check BlockSketch, and follow the two rules below when exploring a sub-tree to search for keyword κ:

(1) **Level-down:** If κ is locally hot at the root of the sub-tree, which is defined as N_{root},the query proceeds one level down to explore both child nodes.
(2) **Jump:** If κ is locally cold at N_{root}, the query retrieves the jump node $\mathcal{J}(\kappa, N_{root})$ and jumps directly to it.

We can determine a keyword is hot by examining the Bloom filter in N_{root} and obtain the jump node by searching the Sketch of it. However, due to the probabilistic nature of the Sketch $N_{root}.\mathcal{S}$, it may return a set $\widehat{\mathcal{J}}$ containing multiple candidate nodes due to false positives, as explained in Sect. 2. In such cases, BlockSketch recursively accesses all nodes in $\widehat{\mathcal{J}}$ to ensure completeness.

10). Otherwise, BlockSketch checks the Bloom filter of N_{root} for keyword κ. A positive result indicates that κ exists in both child nodes' sets, prompting a recursive exploration of the child nodes (lines 12 to 13). If the result is negative, κ resides in at most one child branch, and BlockSketch attempts to jump directly to the next relevant node. It first checks whether the buffer $\mathrm{Buf}(N_{root})$, which temporarily

```
   Input: Key κ, root nodes R, interval [s, t]
   Output: Result set B̂
 1: B̂ ← ∅
 2: for all N_root ∈ R do
 3:     B̂ ← B̂ ∪ Query(κ, N_root, [s, t])
 4: end for
 5: return B̂
 6: function QUERY(κ, N_root, [s, t])
 7:     B̂ ← ∅
 8:     if time_intersect(N_root.T, [s, t]) is false then
 9:         return B̂
10:     end if
11:     if Positive(N.F, κ) then                          ▷ Level-down rule
12:         B̂ ← B̂ ∪ Query(κ, left(N_root), [s, t])
13:         B̂ ← B̂ ∪ Query(κ, right(N_root), [s, t])
14:     else                                              ▷ Jump rule
15:         if Buf(N_root) has not been released then
16:             J ← get_jump_node(Buf(N_root))
17:             B̂ ← B̂ ∪ Jump_Explore({J}, κ, [s, t])
18:         else
19:             Ĵ ← search_sketch(N_root.S, κ)
20:             B̂ ← B̂ ∪ Jump_Explore(Ĵ, κ, [s, t])
21:         end if
22:     end if
23:     return B̂
24: end function
25: function JUMP_EXPLORE(Ĵ, κ, [s, t])
26:     B̂ ← ∅
27:     for all N ∈ Ĵ do
28:         if N refers to a leaf node then
29:             if time_intersect(N.T, [s, t]) is true then
30:                 B̂ ← B̂ ∪ {N}
31:             end if
32:         else
33:             B̂ ← B̂ ∪ Query(κ, N, [s, t])
34:         end if
35:     end for
36:     return B̂
37: end function
```

Algorithm 2 Intra-window query of BlockSketch

Algorithm 2 details the complete intra-window query process for BlockSketch. The algorithm takes as input the target keyword κ, the set $\mathcal{R}$ of sub-tree roots, and the interval [s, t]. Notably, if BlockSketch is fully constructed, $\mathcal{R}$ contains exactly one root. The query is executed on each sub-tree rooted at every $N_{root} \in \mathcal{R}$, and the result of each sub-query is aggregated into $\widehat{\mathcal{B}}$ (lines 1 to 5). The query process for each sub-tree with root node N_{root} is implemented recursively, as described in lines 6 to 24.

In each iteration, if the interval $N_{root}.T$ and [s, t] do not overlap, the search for that sub-tree terminates (lines 8 to

records jump nodes during the construction, is still available. If the buffer exists, BlockSketch retrieves the accurate jump node for κ. If the buffer is unavailable, BlockSketch queries the Sketch $N_{root}.\mathcal{S}$ to estimate the jump nodes $\widehat{\mathcal{J}}$ for κ (lines 15 to 21). The search then proceeds by jumping to the nodes in $\widehat{\mathcal{J}}$ and exploring them recursively (lines 25 to 37). For each node N, if it is a leaf node within the interval [s, t], BlockSketch adds it to the result set (lines 29 to 31); otherwise, it continues querying the sub-tree rooted at N (lines 33).

5.2 Inter-Window Query Processing

Given the total cache capacity C and the capacity of each window $|W|$, the system supports up to $\lceil C/|W| eil$ windows, denoted as $W_1, \ldots, W_{\lceil C/|W| eil}$, each corresponding to a BlockSketch structure. For a query $M_\kappa^{[s,t]}$, where the interval $[s, t]$ may span multiple windows, the process begins by identifying the windows $W_x, \ldots, W_y$ that overlap with the interval. For each window W_z with $x \leq z \leq y$, the query is executed on the corresponding BlockSketch, resulting in a set of blocks $\widehat{\mathcal{B}}_z$. However, due to the inherent false positives of the Sketch or Bloom filter, $\widehat{\mathcal{B}}_z$ may include unrelated leaf nodes. To ensure accuracy, each block referenced in $\widehat{\mathcal{B}}_z$ is retrieved and verified, refining the results. Finally, the confirmed results from all queried windows are then aggregated and returned as the output.

5.3 False Negative Handling

The current implementation of BlockSketch is vulnerable to false negatives, where the valid results may be omitted due to the Bloom filter's false positive nature. Figure 5 illustrates this issue, where the blue arrows represent the expected search path when querying keyword κ. In this example, κ is cold in N_0^3, and the Sketch of N_0^3 records the jump node $\mathcal{J}(\kappa, N_0^3) = N_2^1$. The Bloom filter of N_2^1 yields a positive result, suggesting that κ is hot in both N_4^0 and N_5^0. However, a false positive in the Bloom filter of N_0^3 causes the search to deviate, as shown by the red arrows. This deviation cannot be corrected since neither the Bloom filter nor the Sketch of N_1^2 encodes κ, according to the optimistic assumption that the search would jump directly from N_0^3 to N_2^1 in our design. To conserve space, κ is not recorded in N_1^2's Sketch. Consequently, the true-positive nodes N_4^0 and N_5^0 are overlooked during the query, leading to incomplete results.

```
1: function QUERY(κ, N_root, [s,t], type, skip)
2:     B̂ ← ∅
3:     if time_intersect(N_root.T, [s,t]) is false then
4:         return B̂
5:     end if
6:     if skip is true then
7:         B̂ ← B̂ ∪ Check_Sketch(κ, N_root, [s,t], type)
8:     else
9:         if Positive(N.F, κ) then
10:            B̂ ← B̂ ∪ Query(κ, left(N_root), [s,t], NJ, false)
11:            B̂ ← B̂ ∪ Query(κ, right(N_root), [s,t], NJ, false)
12:        else
13:            if Buf(N_root) has not been released then
14:                J ← get_jump_node(Buf(N_root))
15:                B̂ ← B̂ ∪ Jump_Explore({J}, κ, [s,t])
16:            else
17:                B̂ ← B̂ ∪ Check_Sketch(κ, N_root, [s,t], type)
18:            end if
19:        end if
20:    end if
21:    return B̂
22: end function

23: function CHECK_SKETCH(κ, N_root, [s,t], type)
24:    B̂ ← ∅
25:    Ĵ ← search_sketch(N_root.S, κ)
26:    if Ĵ = ∅ and type = NJ then                    ▷ Re-entry rule
27:        B̂ ← B̂ ∪ Query(κ, parent(N_root), [s,t], NJ, true)
28:    else
29:        B̂ ← B̂ ∪ Jump_Explore(Ĵ, κ, [s,t])
30:    end if
31:    return B̂
32: end function

33: function JUMP_EXPLORE(Ĵ, κ, [s,t])
34:    B̂ ← ∅
35:    for all N ∈ Ĵ do
36:        if N refers to a leaf node then
37:            if time_intersect(N.T, [s,t]) is true then
38:                B̂ ← B̂ ∪ {N}
39:            end if
40:        else
41:            B̂ ← B̂ ∪ Query(κ, N, [s,t], J, false)
42:        end if
43:    end for
44:    return B̂
45: end function
```

Algorithm 3 Intra-window query with re-entry

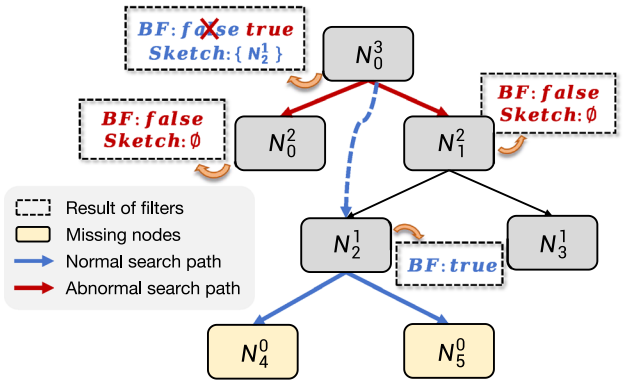


Fig. 5 An example of BlockSketch's false negative

To address this issue, we enhance Algorithm 2 by enabling the search to re-enter the parent node and verify its Sketch when encountering potential false positives. Specifically, in the example scenario, the search revisits N_0^3 to check its Sketch. When examining the Sketch of N_0^2, if the result is empty, the search evaluates whether N_0^2 was reached from its parent N_0^3, indicating that the Bloom filter in N_0^3 returned a positive result for κ. If so, it suggests a false positive in N_0^3's Bloom filter. In such cases, BlockSketch re-enters N_0^3 and bypasses the Bloom filter check, ensuring the correct search path is followed. Therefore, we add a new rule when exploring nodes.

(3) **Re-entry:** If node N_{root} is explored using the level-down rule and further exploration cannot be performed through the Bloom filter and Sketch of N_{root}, the query re-enters the parent of N_{root} and directly checks its Sketch without examining the Bloom filter.

Algorithm 3 outlines the query process with re-entry, highlighting the enhancements in blue for clarity while omitting other details for simplicity. The function Query includes two additional parameters: *type* and *skip*. The *type* parameter can be either J (jump) or NJ (not jump), indicating whether the search reaches N_{root} through a jump. The *skip* parameter is a flag specifying whether to bypass the Bloom filter check during re-entry. If the Sketch's search result is empty and the *type* is NJ, suggesting a false positive in the higher-level Bloom filter, the query re-enters the parent node of N_{root} (lines 26 to 27). The parameter NJ in line 27 specifies that the re-entry into the parent node follows the same path as the original traversal. During re-entry, the Bloom filter check is skipped, and the search proceeds directly to examine the Sketch (lines 6 to 7). Note that although the probability is extremely low if two consecutive Bloom filter checks both result in false positives, our re-entry strategy ensures that the query can always backtrack to the parent node of the first Bloom filter that generated a false positive.

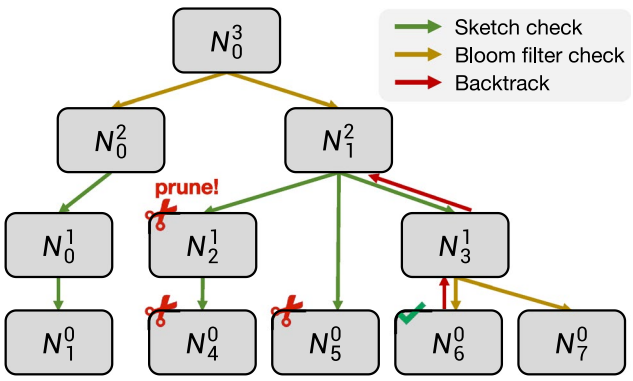


Fig. 6 An example of tree-based pruning

5.4 Optimization

So far, we have developed a fast and space-efficient sketch structure to support keyword search in various blockchain data analysis scenarios. However, there remains room to further optimize both query efficiency and accuracy.

5.4.1 Flat Node

In BlockSketch, each index node is equipped with a Bloom filter and a Sketch to encode keywords from its descendant leaf nodes. At the lower levels of the tree, index nodes typically cover only two or four blocks and manage a limited number of keywords. When the queried keyword exhibits high multiplicity, a greater number of low-level index nodes are inspected, resulting in significantly increased query latency. To further reduce query latency, we introduce a more efficient approach by enabling multiple nodes to share a single Sketch. Provided that the total number of keys across these nodes remains within a reasonable range, the accuracy of a shared Sketch is nearly equivalent to that of multiple smaller Sketches, while significantly reducing the number of nodes to be checked. Specifically, we merge a set of τ leaf node into a compact structure known as a "flat node", which represents a compressed subtree from the original BlockSketch. This flat node is equipped with a single shared Sketch, thereby achieving enhanced query performance with uncompromised space efficiency.

Given a set of τ leaf nodes $\mathcal{L} = \{N_0, \ldots, N_{\tau-1}\}$, the construction of a flat node N_{flat} is outlined in Algorithm 4. First, the hot and cold keywords within $\mathcal{L}$ are determined (lines 3 to 4). Unlike standard nodes, a keyword κ is classified as hot if it appears in the sets of at least two leaf nodes within $\mathcal{L}$; otherwise, it is categorized as cold. Subsequently, the buffer of N_{flat} is updated to record the jump nodes for all keywords concerning N_{flat} (lines 5 to 7). Since a flat node encompasses more than two child nodes, a Sketch is utilized in N_{flat} to identify the leaf nodes of each keyword (lines 8 to 9).

```
function CREATE_FLAT_NODE(L)
    N_flat ← new empty flat node
    H(N_flat) ← hot keywords in Set(L) = ⋃_{N_i ∈ L} Set(N_i)
    C(N_flat) ← cold keywords in Set(L)                          ▷ Set(L) \ H(N_flat)
    buf ← {⟨κ, N_i⟩ ∈ Buf(N_i) | κ ∈ C(N_flat) ∧ N_i ∈ L}
    buf ← buf ∪ {⟨κ, N_flat⟩ | κ ∈ H(N_flat)}
    set_buf(N, buf)
    S ← {⟨κ, N_i⟩ ∈ Buf(N_i) | κ ∈ H(N_flat) ∧ N_i ∈ L}
    insert_sketch(N_flat.S, S)
    return N_flat
end function
function SEARCH_FLAT_NODE(κ, N_flat, [s, t])
    if time_intersect(N_flat.T, [s, t]) is false then
        return
    end if
    J ← search_sketch(N_flat.S, κ)
    for all N ∈ J do
        if time_within(N.T, [s, t]) is true then
            B ← B ∪ {N}
        end if
    end for
end function
```

Algorithm 4 Maintenance of flat node

The function Search_Flat_Node() defines the search procedure within the flat node N_{flat}. Initially, if the interval $N_{flat}.T$ does not overlap with $[s, t]$, the search terminates. Otherwise, the Sketch is employed to retrieve all candidate blocks (or leaf nodes) where κ may exist, followed by a verification of the interval. Importantly, a flat node does not require a Bloom filter for hot keywords, as the Sketch alone is sufficient to identify the target blocks. Unlike hot keywords, cold keywords in $\mathcal{L}$ are encoded in Sketches of higher-level nodes when they become hot during construction.

5.4.2 Pruning Strategy

The construction process of BlockSketch reveals that each Sketch query result contains exactly one leaf node—either directly or indirectly—that holds the true-positive block, based on which we design a fine-grained pruning strategy to further enhance query accuracy. We explore the inter-block relationship within the result set and prioritize checks on blocks, thereby filtering out the blocks which do not meet the query constraint. The query process is formally modelled as a k-ary search tree, as illustrated in Fig. 6, where only the nodes visited during the query are included in the tree. Without pruning, when BlockSketch returns a set of candidate blocks, all of these blocks are randomly checked one by one to determine the true-positive blocks. This approach disregards dependencies between blocks. Since not all the blocks in the result set contain the transaction satisfying the query constraint, when one is confirmed as true-positive, others exhibit high false-positive likelihoods, enabling pruning to eliminate redundant checks.

When tree-based pruning is activated, block verification prioritizes leaf nodes with the longest search path. If a leaf node l_1 is validated as true-positive, the algorithm recursively ascends the search path to trace all index nodes added by Sketches until reaching the root. For each node along the path, its sibling nodes (i.e., other child nodes of its parent) and the corresponding sub-trees are pruned, as l_1's confirmation renders further checks redundant. Taking Fig. 6 as an example, when the leaf node N_6^0 is first checked and it contains the target transaction, we ascend its search path and identify prunable nodes N_5^0 and N_2^2. Since N_2^2 is an index node, all nodes in its sub-tree (i.e., $\{N_4^0\}$) must be pruned according to the strategy. This approach reduces the search space from 5 blocks to 3 blocks, eliminating redundant checks while maintaining query completeness. Although this strategy is effective theoretically, it may fail in rare cases. For example, suppose l_1 has a sibling node l_2, which is a false positive from its parent's check but contains the target transaction. If l_2 has a higher check priority, l_1 will be pruned incorrectly, thereby resulting in the final false positive. To mitigate this, we add an additional rule: if there exists another leaf node l_2' such that $l_2 = l_2'$, pruning is aborted to ensure no true-positive nodes are excluded.

5.5 Analysis

We denote the false positive rates of the Bloom filter and the Sketch in BlockSketch as ϵ_b and ϵ_s, respectively. Although the number of keywords encoded in the filter varies across different levels, we ensure that the values of ϵ_b and ϵ_s remain within a small range, respectively, by adjusting the filter parameters during initialization.

When a Sketch of Node N_{root} at level u performs a membership query for a keyword κ, it returns a set of jump nodes spanning levels 0 to $u-1$. We define p_l

as the probability that a node is located at level l, where $p_0 \gg p_1 \gg \ldots \gg p_{u-1}$ and $\sum_{l=0}^{u-1} p_l = 1$. Since only one candidate node returned by the Sketch is the actual jump node $\mathcal{J}(\kappa, N_{root})$, the expected number of remaining false-positive nodes at level l is:

$$E_l = \epsilon_s \cdot p_l/(1-\epsilon_s).$$

Given that the number of nodes at level l in N_{root}'s sub-tree is bounded by 2^{u-l}, checks on nodes at the same level may be duplicated. However, since ϵ_s is typically low, especially when querying keywords with low multiplicity. Consequently, the maximum number of distinct checks for nodes at level l resulting from false positives does not exceed 2^{u-l}.

Among the nodes returned by the Sketch, leaf nodes are directly added to the result set $\widehat{\mathcal{B}}$, with an expected number of $\epsilon_s \cdot p_0/(1-\epsilon_s)$. For an internal node N, its Bloom filter is first checked to determine whether κ is hot in both child nodes. Since this check process is unnecessary and triggered by a false positive, the Bloom filter is expected to return a negative result. As a result, the sub-tree of N can be successfully pruned, preventing false-positive blocks from being added to $\widehat{\mathcal{B}}$. However, if the Bloom filter returns a positive result, both of N's child nodes will be examined with a probability of ϵ_b. Let F_N denote the number of false-positive blocks resulting from checking N. Then, F_N can be expressed as:

$$F_N = \epsilon_b \cdot (F_{\text{left}(N)} + F_{\text{right}(N)}).$$

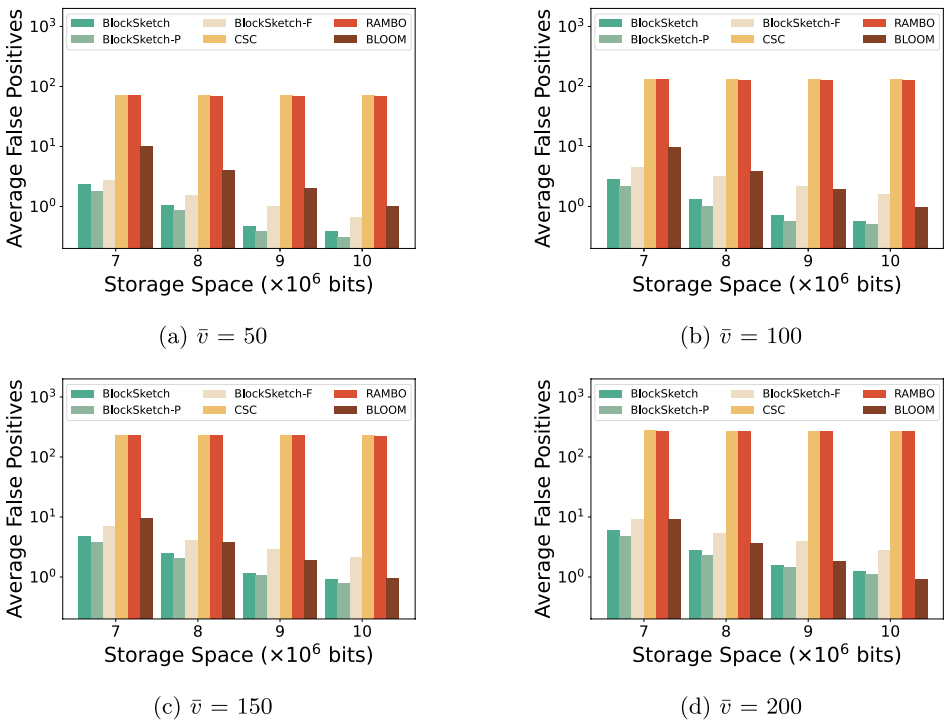
Once the Bloom filter of a node at level 1 is checked and a false positive occurs, two corresponding blocks will be added to $\widehat{\mathcal{B}}$. Consequently, we can further represent F_N as:

$$F_N = 2^l \epsilon_b^l.$$

Given the distribution of nodes obtained from the Sketch across level 0 to $u-1$, the total number of false-positive blocks introduced into $\widehat{\mathcal{B}}$ due to false positives in the Sketch is:

$$\sum_{l=0}^{u-1} \epsilon_s \cdot p_l/(1-\epsilon_s) \cdot 2^l \epsilon_b^l.$$

In summary, BlockSketch minimizes the impact of nodes not along the normal search path on the result set by using additional Bloom filter checks for each internal node returned by a Sketch query, compared to other single-Sketch solutions.



(a) $\bar{v} = 50$ (b) $\bar{v} = 100$

(c) $\bar{v} = 150$ (d) $\bar{v} = 200$

Fig. 7 Average false positives with varying storage space

6 Evaluation

In this section, we first present the configuration of the experiments, and then compare BlockSketch with existing methods to demonstrate the effectiveness and efficiency of the proposed approach.

6.1 Experiment Setup

We implement BlockSketch using Golang 1.22.2, comprising approximately 4000 lines of core project code. For the Bloom filter, we use the third-party Go library `bitset`[3] to construct its underlying data structure and the `xxhash`[4] library for hash functions with different seeds used during keyword encoding. These libraries are also utilized in the implementation of the Sketch. Another critical component of BlockSketch is the buffer, as described in Sect. 4.2. Since the buffer is maintained temporarily during tree construction, we use a map to record the jump nodes of all keywords, which can also be utilized to accelerate the Set() operation.

6.1.1 Comparisons

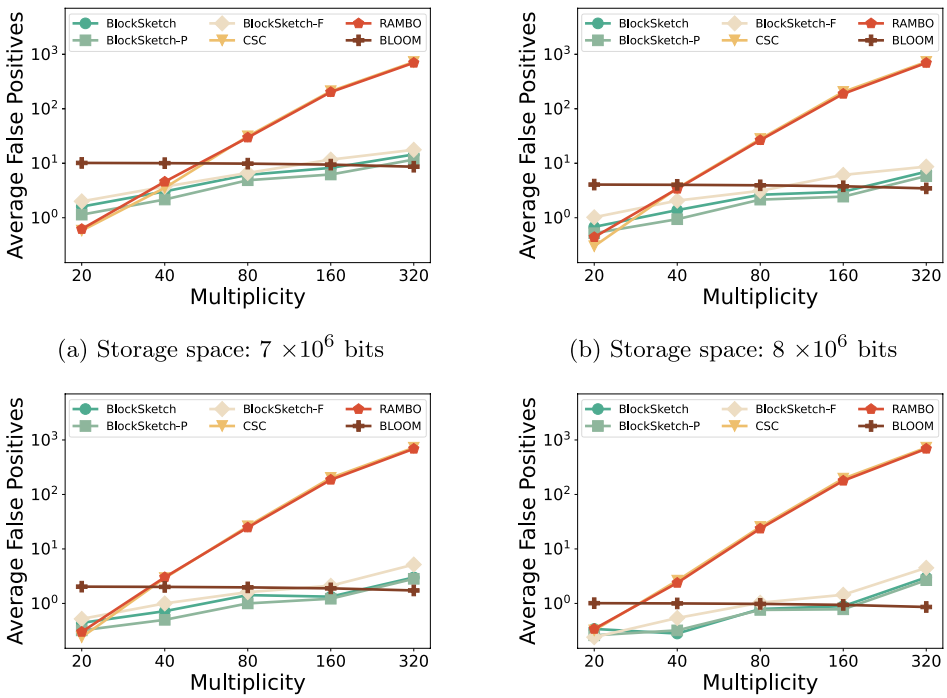
The methods compared can be classified as follows:

- **BLOOM**: BLOOM refers to a design that assigns a dedicated Bloom filter to each block, requiring queries to traverse every individual filter, which incurs a time complexity of $O(n)$.
- **RAMBO and CSC**: These methods are based on the Count-Min Sketch, and facilitate membership query for a key across multiple sets. RAMBO [25] utilizes the Bloom filter as its basic filter unit, while CSC [22] additionally offers the implementation using the Cuckoo filter [24].
- **BlockSketch**: BlockSketch denotes the native implementation of our proposed method. BlockSketch-F refers to the optimized variant introducing flat nodes into BlockSketch, while BlockSketch-P indicates the version incorporating a pruning strategy during query execution.
- **BloomTree**: BloomTree [26], similar to the rough solution discussed in Sect. 3.2, merges multiple Bloom filters into a binary search tree, reducing the time complexity of membership queries from $O(n)$ to $O(\log n)$.

To ensure a fair comparison, we reproduce CSC and RAMBO in Go. Additionally, CSC is integrated as the Sketch component within the nodes of BlockSketch. For both CSC and RAMBO, we adjust certain parameters to control the number of false positives during testing, while keeping most other settings close to those recommended in their original papers. Minor modifications are made to achieve comparable space usage across different methods.

[3] https://github.com/bits-and-blooms/bitset.

[4] https://github.com/cespare/xxhash.



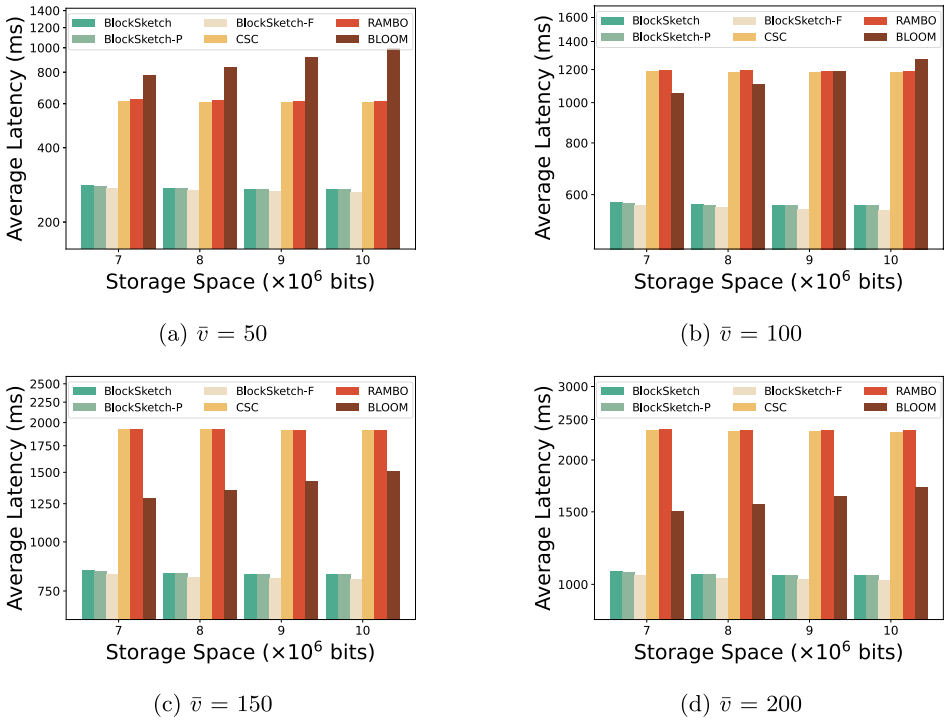
(a) Storage space: 7 $\times 10^6$ bits

(b) Storage space: 8 $\times 10^6$ bits

(c) Storage space: 9 $\times 10^6$ bits

(d) Storage space: 10 $\times 10^6$ bits

Fig. 8 Average false positives with varying multiplicities



(a) $\bar{v} = 50$ (b) $\bar{v} = 100$

(c) $\bar{v} = 150$ (d) $\bar{v} = 200$

Fig. 9 Average query latency with varying storage space

6.1.2 Metrics

To comprehensively evaluate the performance of BlockSketch and the compared methods, we use the following metrics and key parameters:

- **Average False Positives**: The average number of false-positive blocks returned per query, which correlates with the useless block verification overhead.
- **Average Latency (ms)**: The average end-to-end time to execute a keyword search, including index traversal, filter checks, and final block verification.
- **Storage Space (bits)**: The total memory space consumed by the constructed index for a given window of blocks.
- **Multiplicity ($\bar{v}$)**: A key workload parameter defined as the number of distinct blocks in which a specific keyword appears within a given window.

6.1.3 Datasets and Workloads

We utilize Ethereum blocks ranging from height 18,000,000 to 19,000,000 and extract the complete transaction data using `Ethereum ETL`[5]. These transactions encompass native Ether transfers, smart contract calls, and other blockchain interactions. All transactions are standardized into the tuple format defined in Sect. 2. The window size $|W|$ is set to 2048. To evaluate the performance of the methods on keyword search, we generate four types of workloads with the average multiplicity set to 50, 100, 150, and 200, respectively.

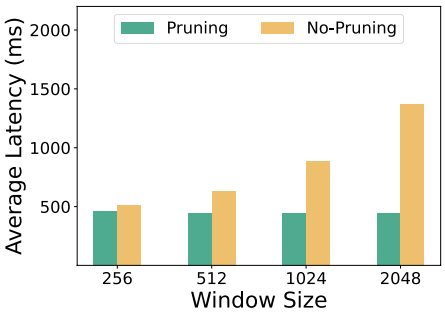
6.1.4 Testbed

We run experiments on a physical machine equipped with a single Intel Xeon Gold 6330 CPU @ 2.00GHz processor featuring 28 physical cores and 56 threads. The machine has a total of 125 GB of DRAM and a 1 Gbps network bandwidth. We set up a full Ethereum node in a local environment using go-ethereum version v1.14.11 to provide real test data. A test chain is launched using geth to validate the accuracy and efficiency of BlockSketch.

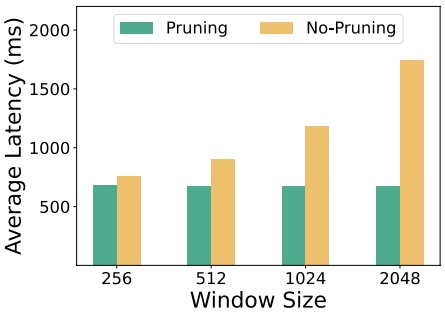
6.2 Query Accuracy

We evaluate the accuracy of all approaches under varying space usage by querying 100 random keywords and counting the false positive cases, as shown in Fig. 7. Overall, allocating more space reduces false positive cases across all structures, though the improvement of CSC and RAMBO is limited due to their design constraints. BlockSketch and its variations consistently outperform CSC and RAMBO. For instance, in Fig. 7d, with 10^7 bits of storage space, the average number of false positives for BlockSketch and BlockSketch-P remains around 1, while CSC and RAMBO exceed 271.23 and 270.91 false positives, respectively. The

[5] https://github.com/blockchain-etl/ethereum-etl.

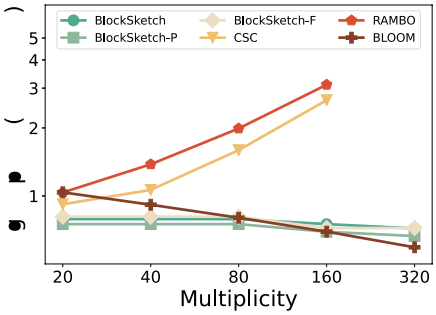


(a) Multiplicity: 40

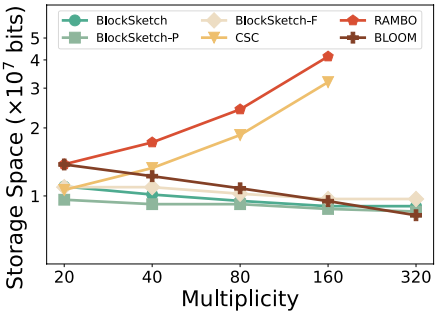


(b) Multiplicity: 80

Fig. 10 Query latency with time pruning



(a) False positive rate: 5%



(b) False positive rate: 1%

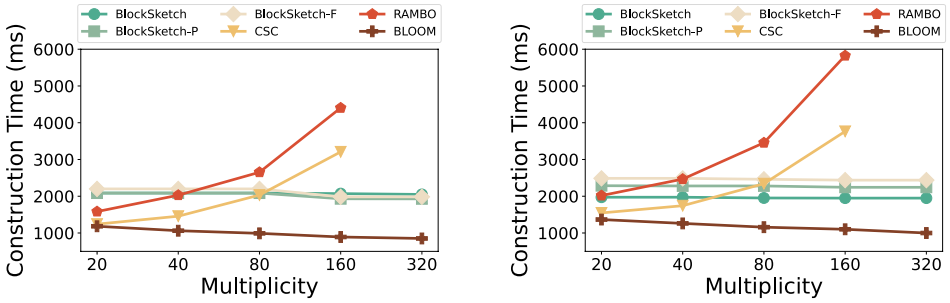
Fig. 11 Storage space with varying multiplicity

false positive rate of BLOOM depends on the configuration of individual Bloom filters and the number of blocks. As multiplicity increases, the number of blocks susceptible to false positives decreases correspondingly, thereby reducing BLOOM's overall false positives. However, this advantage over BlockSketch only manifests under the storage space size of 10^7 bits, as demonstrated in Fig. 7d. Notably, BlockSketch-F reduces latency by flattening lower-level subtrees into compressed nodes at the cost of marginal query accuracy degradation, which will be analyzed later. The improvement in accuracy arises from two factors. First, BlockSketch encodes only a limited number of keywords at each node, enhancing the accuracy of both Bloom filters and Sketches. Second, its hierarchical structure prunes false positives from intermediate query results. Furthermore, BlockSketch-P further outperforms BlockSketch by incorporating the pruning strategy, filtering more irrelevant blocks to be added to the result set. As the hot keyword rate increases, the false positive cases rise for all structures due to the higher multiplicity of keywords, which increases the false positive rate of Sketches. However, BlockSketch and BlockSketch-P are the least affected. For example, when the average multiplicity increases from 50 to 200, the false positive cases for BlockSketch-P grow modestly from 0.31 to 1.11, compared to a rise from 68.8 to 270.91 for RAMBO.

Furthermore, we allocate the same amount of storage space to each method while varying the multiplicity of queried keywords. Figure 8 shows the false positive cases with varying multiplicity under different space configurations. Overall, BlockSketch and its variants outperform the compared methods in most cases. When $v = 20$, the average number of false positives of BlockSketch is 44.56% of the mean value of the baselines. As v increases, this ratio gradually decreases to 11.38% when $v = 80$, and further to 1.52% when $v = 320$. Consistent with previous results, increasing space allocation reduces false positives across all the structures except BLOOM, whose false positives are hardly affected by multiplicity. As is shown in Fig. 8a, CSC and RAMBO achieve near-perfect accuracy (around 0.5) when the multiplicity is 20, slightly outperforming BlockSketch and its variations, which exhibit between 1 and 2. This is because BlockSketch allocates less space to the Sketches, resulting in a slightly higher false positive rate for low-multiplicity queries. However, as keyword multiplicity increases, the false positive rates of CSC and RAMBO rise sharply, surpassing those of BlockSketch when the multiplicity is around 30. The superior performance of BlockSketch and its variations in these cases is attributable to their use of multiple Bloom filters in the query process, which effectively filters out many false positives that the Sketch would otherwise introduce.

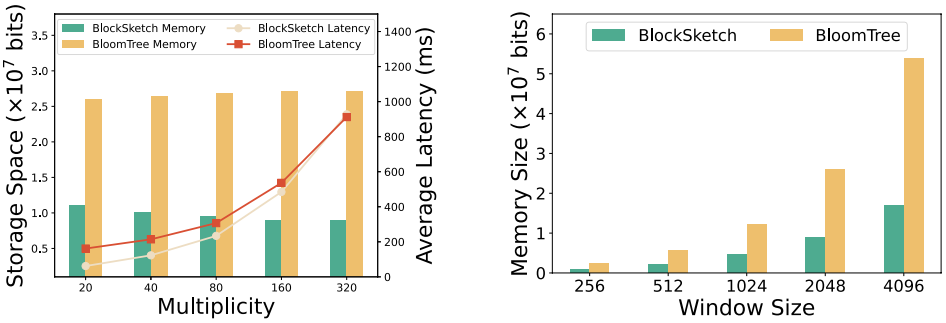
6.3 Query Efficiency

We evaluate the query efficiency of different structures by measuring the average latency of a single keyword search



(a) False positive rate: 5% (b) False positive rate: 1%

Fig. 12 Construction time with varying multiplicity



(a) Storage space and latency (b) Scalability

Fig. 13 Comparison with BloomTree

under various workloads, as shown in Fig. 9. In general, the query latency of each structure increases with the average multiplicity since it needs more operations to identify a larger number of results. For example, when the average multiplicity rises from 50 to 100 (shown in Fig. 9a and Fig. 9b), query latency nearly doubles for all structures at a storage space size of 7×10^6 bits.

BLOOM exhibits the highest latency under an average multiplicity of 50, followed by RAMBO and CSC, due to its requirement to uniformly inspect each block's filter. As multiplicity increases, the number of blocks requiring verification in BLOOM becomes significantly lower than that of CSC and RAMBO, resulting in reduced query latency. For instance, with an equivalent space allocation of 10^7 bits, BLOOM's latency at $\bar{v} = 50$ is $1.62\times$ that of RAMBO, whereas at $\bar{v} = 150$, it becomes $0.79\times$ that of RAMBO. In contrast, BlockSketch maintains consistently lower query latency by avoiding traversal of all nodes during queries while achieving superior accuracy. Under optimal conditions, it achieves latency reductions of 56%, 57%, and 73% compared to CSC, RAMBO, and BLOOM, respectively. The enhanced variant BlockSketch-P further reduces overall latency through improved query precision, whereas BlockSketch-F condenses multi-node subtree checks into a single verification step, delivering up to a 10% performance improvement over the base BlockSketch implementation.

Notably, as space allocation increases, all structures except BLOOM exhibit varying degrees of latency reduction. While increased space allocation extends disk I/O time for loading filters, the reduction in false positive rates leads to decreased I/O overhead for block verification, which dominates the overall latency change. Among them, CSC and RAMBO demonstrate more limited latency improvements. For example, at $\bar{v} = 200$, CSC's latency decreases marginally from 2353 ms to 2334 ms when storage space grows from 7×10^6 bits to 10^7 bits, as its design principles prevent space increments from significantly enhancing query precision. In contrast, BlockSketch achieves a latency reduction from 1075 ms to 1051 ms under the same conditions, indicating that additional space allocation empowers BlockSketch with stronger false positive filtering capabilities.

6.4 Effectiveness of Time Pruning

Figure 10 illustrates the effectiveness of time pruning in the query process by preventing the exploration of nodes outside the specified interval. Without time pruning, the search must traverse the entire window. We construct two test workloads in which each key has a multiplicity of 40 or 80 within the window of size 256, respectively.

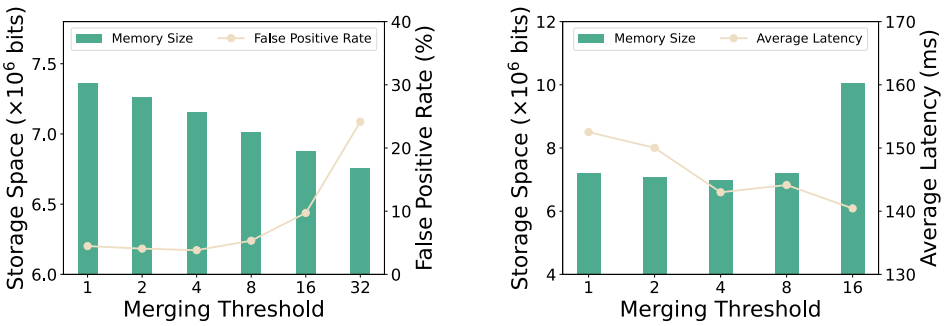


Fig. 14 Impact of varying τ on BlockSketch

(a) Uniform parameter setting

(b) False positive rate: 1%

We further test both scenarios with larger window sizes. As shown in Fig. 10b, the pruning method provides a 1.12× speed boost at window size 256. When using bigger windows without pruning, the system has to check more BlockSketch nodes, causing higher I/O costs for loading filters and slower queries. On the other hand, queries using pruning only check necessary BlockSketch nodes no matter the window size, keeping overall response times stable with only small variations caused by occasional false positives. The speed improvement grows to 2.58× when the window size reaches 2048. However, since loading blocks takes up a large part of the total delay, the speed gains don't increase evenly. When more time is spent checking filters (as measured in Fig. 10a), the benefits of pruning become much clearer, reaching up to 3.09× faster than not using pruning.

6.5 Storage Efficiency

To visually contrast the storage efficiency across different methods, we construct the structures based on two preset false positive rates and measure their required storage space sizes under varying multiplicity. Overall, higher query accuracy demands necessitate additional space allocations for all structures. As multiplicity increases, both CSC and RAMBO require greater space allocations to achieve preset thresholds. As shown in Fig. 11b, CSC's storage consumption rises from 1.06×10^7 bits at multiplicity 20 to 3.19×10^7 bits at multiplicity 160, while RAMBO's space usage remains 1.3× that of CSC in both scenarios. Notably, Fig. 11 omits CSC and RAMBO's data at multiplicity 320 due to their excessively high space requirements compared to other methods. In contrast, BlockSketch and BLOOM exhibit inverse trends, where their required storage consumption decreases as multiplicity increases. BlockSketch achieves this due to its mechanism of inspecting more Bloom filters to ensure accuracy when processing high-multiplicity queries, whereas BLOOM's reduction stems from fewer false-positive blocks resulting from increased true-positive ones. Between multiplicity levels of 20 to 160, BlockSketch and its variants consistently demonstrate superior space efficiency compared to BLOOM, which aligns with the observations in Fig. 7.

6.6 Construction Efficiency

The construction efficiency of an index is also a critical metric for its practical applicability, so we conduct an evaluation using the same parameters as in the storage overhead comparison. The results are presented in Fig. 12. The BLOOM scheme, which involves only Bloom filters, is a straightforward structure to maintain and thus exhibits the lowest construction time among all methods. For RAMBO and CSC, their simpler structural designs, compared with that of BlockSketch, lead to faster construction times when the multiplicity is low (e.g., 20 and 40). However, as the multiplicity increases to 80 and beyond, their construction times rise sharply, becoming 2.99× and 1.93× longer than that of BlockSketch, as shown in Fig. 12b, respectively, making them the most time-consuming approaches. This is because, to achieve the target accuracy, they must be configured with significantly larger Bloom filters and more repetitions, which leads to the prohibitive storage overhead we observed previously and, in turn, directly impacts their construction time.

In contrast, the construction time of BlockSketch and its variants remains relatively stable despite their more complex structure. As shown in Fig. 12a, as the multiplicity increases from 20 to 320, the construction time changes only slightly from 1973 ms to 1948 ms. This stability stems from two key advantages. First, the differentiated encoding strategy, which handles hot and cold keywords separately, effectively avoids the data redundancy inherent in a uniform encoding scheme. Second, the tree structure of BlockSketch amortizes the impact of potential false positives across multiple levels of checks. This allows us to meet the target accuracy by making only minor parameter adjustments in response to changing multiplicity, unlike RAMBO and CSC, which must allocate substantial extra space to compensate for their accuracy deficiencies. These space savings directly translate to reductions in construction time. Specifically, among

BlockSketch and its two optimized versions, the construction times of BlockSketch and BlockSketch-P are relatively similar because they share an identical structure. Although BlockSketch-P can achieve the same accuracy with less space due to its query-time pruning strategy, this smaller space may lead to a higher rate of hash collisions during Sketch's encoding process, introducing some additional latency. The optimization in BlockSketch-F is structural; while flat nodes make queries on lower levels of the tree more efficient, they introduce an additional overhead during construction, making it more time-consuming compared to the base BlockSketch and BlockSketch-P.

6.7 Comparison with BloomTree

Figure 13 compares the performance of BlockSketch with BloomTree, another tree-based PDS. Parameters are manually adjusted to ensure both approaches achieve a comparable false positive rate. We evaluate their storage consumption and query latency under varying multiplicity. As shown in Fig. 13a, when both methods maintain a 1% false positive rate, BlockSketch achieves storage savings of 57.51%–66.83% due to its differentiated encoding strategy. In comparison, BloomTree requires redundant encoding across every layer of its structure for a single data record. The efficiency gap widens with higher multiplicity levels, as more filters are activated to eliminate false positives. Additionally, BlockSketch reduces query latency by up to 61.77%, primarily because BloomTree's latency stems heavily from loading oversized filters from the storage engine. However, as multiplicity increases, BlockSketch's latency advantage diminishes since its checking process involves more filters and requires computationally intensive sketch checks compared to simple Bloom filters.

Figure 13b demonstrates the scalability of BlockSketch and Bloom-Tree under varying window size, i.e., number of blocks. We construct both structures using the same dataset and ensure their average false positive rates remain around 1% for a fair comparison of space usage. At window size 256, BloomTree requires 2.61× more storage space than BlockSketch. As window sizes expand, both methods exhibit growing storage consumption, but BloomTree's consumption grows at a faster rate. Notably, at window size 4096, BloomTree consumes 3.18× the storage space compared to BlockSketch, demonstrating significantly weaker scalability under large-scale window conditions. This disparity arises because higher-level nodes in BloomTree require larger Bloom filters to accommodate more keywords from a broader range of blocks. In contrast, BlockSketch's storage overhead is determined by the multiplicity distribution of the keywords, avoiding redundant encodings and maintaining efficient space usage.

6.8 Analysis of Merging Threshold τ

BlockSketch+ introduces flat nodes by merging τ consecutive leaf nodes, thereby reducing the number of index nodes and enhancing space efficiency. To determine the optimal value of τ, we conduct queries under consistent workloads, with results shown in Fig. 14a. As τ increases, space usage steadily decreases. This trend occurs because lower-level filter units, primarily Sketches, are underutilized with smaller values of τ. Increasing τ allows Sketches to encode more keywords without expanding, resulting in significant space savings. Additionally, since the Sketch reduces false positives by intersecting the query results from multiple filter groups, when τ is relatively small ($\tau < 8$), the keyword multiplicity within each flat node also remains low, so the intersection remains effective and the false positive rate stays relatively stable. However, as τ continues to increase, some Sketches must expand to accommodate additional keywords, slowing space savings and eventually reaching a plateau. At this stage, each filter group produces a large number of candidate sets, making the intersection operation much less effective. As a result, the false positive rate increases, which reflects the inherent limitation of the Sketch.

In Fig. 14b, we adjust τ and evaluate both storage consumption and query latency. As τ increases, additional space is allocated to the Bloom filters and Sketches within the tree while maintaining a false positive rate of approximately 1%. When $\tau = 4$, the false positive rate remains nearly identical to the default parameter (see Fig. 14a), eliminating the need to adjust other parameters. For other values of τ, we adjust the parameters of Bloom filters and Sketches to ensure the false positive rate remains constant. The results show that overall storage consumption and query latency initially decrease with increasing τ before starting to rise. By considering space usage, query latency, and accuracy, selecting τ values of 4 or 8 achieves an optimal balance.

7 Conclusion

To support efficient keyword search in blockchain systems, we introduce BlockSketch, a novel probabilistic data structure designed to provide high-accuracy queries while minimizing space overhead. BlockSketch employs a binary search tree architecture, with each node incorporating a hybrid filter with Bloom filter and Sketch. BlockSketch

adopts a differentiated encoding strategy that categorizes data into "cold" and "hot" based on their frequency, encoding them into the Sketch and Bloom filter, respectively. Through dynamic space configuration, BlockSketch achieves a balance between query latency, accuracy, and storage efficiency. Additionally, we reduce the query latency of BlockSketch by compacting low-level sub-trees into single flat nodes and reducing checks on false-positive blocks using a tree-based pruning strategy. Extensive experiments validate the query accuracy and space efficiency of the proposed approach.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments and suggestions, which have greatly improved the quality of this manuscript. We also acknowledge the collective efforts of all authors during the development of this work, including their contributions to discussions, implementation, and manuscript refinement.

Author contributions All authors contributed to the conception and design of the study. Yushi Liu conducted the coding, data collection, and analysis. Yushi Liu and Xiaodong Qi drafted the initial version of the manuscript. All authors reviewed and revised previous versions of the manuscript and approved the final version for submission.

Funding This work was supported by the National Key Research and Development Program of China (No. 2023YFC3304700), the Shanghai "Science and Technology Innovation Action Plan" Project (No. 23511100700), and Program of Shanghai Academic/Technology Research Leader (No. 23XD1401100), and the Nanyang Technological University Centre for Computational Technologies in Finance (NTU-CCTF). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NTU-CCTF.

Data Availability The datasets generated and/or analyzed during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests The authors have no relevant financial or non-financial interests to disclose.

Ethics approval and consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Nakamoto S (2008) Bitcoin: A peer-to-peer electronic cash system. Decentralized Business Review
2. Buterin V et al (2013) Ethereum white paper GitHub repository 1:22–23
3. Zetzsche DA, Arner DW, Buckley RP (2020) Decentralized finance. J Financ Regul 6(2):172–203
4. Bartoletti M, Chiang JH-y, Lafuente AL (2021) Sok: lending pools in decentralized finance. In: Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25, 553–578
5. Zhou L, Qin K, Torres CF, Le DV, Gervais A (2021) High-frequency trading on decentralized on-chain exchanges. In: 2021 IEEE Symposium on Security and Privacy (SP), 428–445
6. Yeoh P (2017) Regulatory issues in blockchain technology. J Financ Regul Compl 25(2):196–208
7. Wu J, Lin D, Fu Q, Yang S, Chen T, Zheng Z, Song B (2023) Toward understanding asset flows in crypto money laundering through the lenses of ethereum heists. IEEE Trans Inf Forensics Secur 19:1994–2009
8. Hristidis V, Papakonstantinou Y (2002) Discover: Keyword search in relational databases. In: VLDB'02: Proceedings of the 28th International Conference on Very Large Databases, 670–681
9. Liu F, Yu C, Meng W, Chowdhury A (2006) Effective keyword search in relational databases. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 563–574
10. Qin L, Yu JX, Chang L (2009) Keyword search in databases: the power of rdbms. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, 681–694
11. Li Y, Zheng K, Yan Y, Liu Q, Zhou X (2017) Etherql: a query layer for blockchain system. In: Database Systems for Advanced Applications: 22nd International Conference, DASFAA 2017, Suzhou, China, March 27-30, 2017, Proceedings, Part II 22, 556–567
12. Zhu Y, Zhang Z, Jin C, Zhou A, Yan Y (2019) Sebdb: Semantics empowered blockchain database. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), 1820–1831
13. Xu, C., Zhang, C., Xu, J (2019) vchain: Enabling verifiable boolean range queries over blockchain databases. In: Proceedings of the 2019 International Conference on Management of Data, 141–158
14. Wang H, Xu C, Zhang C, Xu J, Peng Z, Pei J (2022) vchain+: Optimizing verifiable blockchain boolean range queries. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE), 1927–1940
15. Zhang C, Xu C, Xu J, Tang Y, Choi B (2019) Gem 2-tree: A gas-efficient structure for authenticated range queries in blockchain. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), 842–853
16. Comer D (1979) Ubiquitous b-tree. ACM Computing Surveys (CSUR) 11(2):121–137
17. Schütze H, Manning CD, Raghavan P (2008) Introduction to Information Retrieval **39**
18. Wu H, Peng Z, Guo S, Yang Y, Xiao B (2021) Vql: efficient and verifiable cloud query services for blockchain systems. IEEE Trans Parallel Distrib Syst 33(6):1393–1406
19. Tong X, Tang H, Jiang N, Fan W, Gao Y, Deng S, Zhang Z, Jin C, Yang Y, Qin G (2021) Sql-middleware: enabling the blockchain with sql. In: Database Systems for Advanced Applications: 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11–14, 2021, Proceedings, Part III 26, 622–626

20. Day A, Medvedev E, Risdal M, Katesit T (2019) Ethereum in bigquery: a public dataset for smart contract analytics. Google Cloud Blog
21. Kalodner H, Möser M, Lee K, Goldfeder S, Plattner M, Chator A, Narayanan A (2020) {BlockSci}: Design and applications of a blockchain analysis platform. In: 29th USENIX Security Symposium (USENIX Security 20), 2721–2738
22. Li R, Wang P, Zhu J, Zhao J, Di J, Yang X, Ye K (2021) Building fast and compact sketches for approximately multi-set multi-membership querying. In: Proceedings of the 2021 International Conference on Management of Data, 1077–1089
23. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. Commun ACM 13(7):422–426
24. Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD (2014) Cuckoo filter: Practically better than bloom. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, 75–88
25. Gupta G, Yan M, Coleman B, Kille B, Elworth RL, Medini T, Treangen T, Shrivastava A (2021) Fast processing and querying of 170tb of genomics data via a repeated and merged bloom filter (rambo). In: Proceedings of the 2021 International Conference on Management of Data, 2226–2234
26. Yoon MK, Son J, Shin S-H (2014) Bloom tree: A search tree based on bloom filters for multiple-set membership testing. In: IEEE INFOCOM 2014-IEEE Conference on Computer Communications, 1429–1437
27. Bradley P, Den Bakker HC, Rocha EP, McVean G, Iqbal Z (2019) Ultrafast search of all deposited bacterial and viral genomic data. Nat Biotechnol 37(2):152–159
28. Guo D, Wu J, Chen H, Yuan Y, Luo X (2009) The dynamic bloom filters. IEEE Trans Knowl Data Eng 22(1):120–133
29. Castro M, Liskov B et al (1999) Practical byzantine fault tolerance In OsDI 99:173–186
30. Liu H, Luo X, Liu H, Xia X (2021) Merkle tree: A fundamental component of blockchains. In: 2021 International Conference on Electronic Information Engineering and Computer Science (EIECS), 556–561
31. Li S, Zhang Z, Xiao J, Zhang M, Yuan Y, Wang G (2024) Authenticated keyword search on large-scale graphs in hybrid-storage blockchains. In: 2024 IEEE 40th International Conference on Data Engineering (ICDE), 1958–1971
32. El-Hindi M, Binnig C, Arasu A, Kossmann D, Ramamurthy R (2019) Blockchaindb: a shared database on blockchains. Proc VLDB Endow 12(11):1597–1609
33. Fan L, Cao P, Almeida J, Broder AZ (2000) Summary cache: a scalable wide-area web cache sharing protocol. IEEE ACM Trans Netw 8(3):281–293
34. Shanmugasundaram K, Brönnimann H, Memon N (2004) Payload attribution via hierarchical bloom filters. In: Proceedings of the 11th ACM Conference on Computer and Communications Security, 31–41
35. Deng F, Rafiei D (2006) Approximately detecting duplicates for streaming data using stable bloom filters. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, 25–36
36. Almeida PS, Baquero C, Preguiça N, Hutchison D (2007) Scalable bloom filters. Inf Process Lett 101(6):255–261
37. Yang T, Liu AX, Shahzad M, Yang D, Fu Q, Xie G, Li X (2017) A shifting framework for set queries. IEEE ACM Trans Netw 25(5):3116–3131
38. Zhang H, Lim H, Leis V, Andersen DG, Kaminsky M, Keeton K, Pavlo A (2018) Surf: Practical range query filtering with fast succinct tries. In: Proceedings of the 2018 International Conference on Management of Data, 323–336
39. Yu M, Fabrikant A, Rexford J (2009) Buffalo: Bloom filter forwarding architecture for large organizations. In: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, 313–324
40. Hao F, Kodialam M, Lakshman T, Song H (2009) Fast multiset membership testing using combinatorial bloom filters. In: IEEE INFOCOM 2009, 513–521
41. Dai H, Zhong Y, Liu AX, Wang W, Li M (2016) Noisy bloom filters for multi-set membership testing. In: Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science, 139–151
42. Cormode G, Muthukrishnan S (2005) An improved data stream summary: the count-min sketch and its applications. J Algorithms 55(1):58–75