STEM^2: A Fast and Space-efficient Data Structure for Exact Multi-Set Membership Queries

Yannian Niu
University of Connecticut
yannian.niu@uconn.edu

Song Han
University of Connecticut
song.han@uconn.edu

Minmei Wang*
University of Connecticut
minmei.wang@uconn.edu

ABSTRACT

Multi-set membership queries are ubiquitous in networking and database systems. Current solutions force a difficult compromise: hash tables guarantee correctness but suffer from high memory footprints, while filter-based approaches optimize space at the cost of probabilistic errors. In this paper, we propose STEM^2, a fast and space-efficient data structure that achieves 100% query accuracy and can support dynamic key updates for multi-set membership queries. STEM^2 utilizes a balanced binary tree architecture where each non-leaf node incorporates a novel Exact Binary Set Separator (XBSS) to partition keys into two disjoint groups. A key innovation of our design is a minimized hashing scheme that requires only two hash computations per key lookup, significantly reducing computational overhead. Additionally, STEM^2 separates the control plane and the data plane: the control plane handles construction and dynamic updates, while the data plane is dedicated to serving efficient membership queries. Extensive experiments show that STEM^2 achieves over 120 million operations per second (Mops) in lookup throughput, outperforming the state-of-the-art Coloring Embedder by 20% and the Ludo hashing by up to 21.6×, while maintaining compact memory cost and exact correctness.

PVLDB Reference Format:
Yannian Niu, Song Han, and Minmei Wang. STEM^2: A Fast and Space-efficient Data Structure for Exact Multi-Set Membership Queries. PVLDB, 19(9): 2426 - 2438, 2026.
doi:10.14778/3819518.3819561

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/YannianNiu/STEM2_VLDB2026.

1 INTRODUCTION

Given a universal key set $U = \{x_1, x_2, ..., x_i, ..., x_n\}$, which is partitioned into m disjoint subsets such that $S_1 \cup S_2 \cup ... \cup S_{m-1} \cup S_m = U$ and $S_i \cap S_j = \emptyset$ for all $i \neq j$, the multi-set membership queries (MS-MQ) is defined as follows: Given a key $x \in U$, determines the set ID $i \in \{1, 2, ..., m\}$ such that $x \in S_i$.

MS-MQ is a fundamental primitive in a wide range of network and database applications, including network traffic management and routing [33, 42, 43], distributed data storage/caching [11, 46], security and privacy-preserving systems [27, 45], and a variety of data mining tasks such as customer purchase analysis [7, 24]. Below, we present two representative case studies.

Case 1: Layer-2 packet forwarding. Packet forwarding [33] is a fundamental network function that directs incoming packets to their corresponding output ports. Specifically, layer-2 switches perform this operation by querying a MAC table. This querying process can be viewed as a MS-MQ problem, where the MAC address of a packet serves as the key and the associated port number represents the set ID. A typical MAC table may contain tens of thousands of entries spanning multiple ports [36]. Efficiently determining the correct port for a given MAC address is therefore essential for high-speed packet processing.

Case 2: Entity tag query. Entity tag query is a common operation in many practical systems, where each entity is assigned exactly one tag (or identifier) from a finite set. For example, customers may be classified into different VIP levels (Bronze, Silver, Gold, Platinum), or IoT devices may be grouped into priority tiers for resource allocation. The tag query can be formulated as a MS-MQ problem, where the entity identifier (e.g., customer ID or device ID) serves as the key, and the corresponding tag (e.g., VIP level or priority tier) represents the set ID.

A desired MS-MQ solution should meet the following properties:

- **High lookup throughput.** Efficiently query the set ID for a given key is essential. High lookup throughput directly impacts the scalability and responsiveness of the applications, enabling real-time query processing even under large-scale workloads.
- **Correctness of query results.** Ensuring 100% accuracy in query results is critical. Although some applications may tolerate a small error rate, even minor inaccuracies can cause significant resource inefficiencies, such as wasted computing power, I/O capacity, and network bandwidth. For instance, an incorrect forwarding decision may send a packet to the wrong port, leading to misdelivery and subsequent retransmissions.
- **Low memory cost.** Devices such as switches, routers, and IoT nodes that perform MS-MQ often operate under strict resource constraints. Designing a compact data structure is essential to enable efficient operation within limited memory budgets.
- **Dynamic update support.** Since set memberships can change over time, the proposed data structure and algorithm should efficiently handle dynamic key updates, including insertions, deletions, and migrations.

Existing solutions for MS-MQ problem can be divided into two categories: hash tables and filter-based methods. Hash tables are conventional data structures for key-value storage and can be adapted for MS-MQ by storing the set ID as the value associated with each key. However, traditional hash tables incur high memory

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819561

cost due to the requirement of explicit key storage. Recently, several efficient key-value stores have been proposed, including partial key Cuckoo hashing [9, 21], SetSep [10, 47], Bloomier/Othello [4, 5, 44], and Ludo hashing [32]. These key-value stores are memory-efficient and can support high lookup throughput. Specifically, Ludo hashing achieves the lowest space cost for dynamic key-value lookups among existing solutions [32]. However, traditional key-value lookup tables are designed for general-purpose applications, where both keys and values may vary arbitrarily. In contrast, MS-MQ involves value drawn from a fixed and finite set (e.g., set identifiers). Moreover, the size distribution of sets in such applications is often highly skewed. For instance, network traffic flows in large-scale systems typically follow Zipfian or Zipfian-like distributions [30]. These unique characteristics make general-purpose hash tables less efficient in both memory usage and lookup performance. Some research work focuses on filter-based methods that trade off memory cost, lookup throughput, and accuracy, allowing small error rates [14, 25, 29, 36]. However, these approaches do not guarantee 100% query accuracy. Consequently, there is a need for an efficient structure and algorithm for MS-MQ that ensure full accuracy while maintaining high performance, low memory cost, and support for dynamic updates.

In this paper, we present a holistic design for MS-MQ, called **S**et **T**ree for **E**xact **M**ulti-set **M**embership Queries (STEM^2). STEM^2 organizes keys from multiple sets in a binary tree. At each internal node, a binary set separator recursively partitions the sets into two disjoint groups until each leaf corresponds to a single set. To support this design, we develop an efficient Exact Binary Set Separator (XBSS), which serves as the core building block of STEM^2.

To reconcile lookup efficiency with update agility, STEM^2 presents a decoupled architecture with separate control and data planes. The control plane design, $\texttt{STEM}^2_\texttt{C}$, is responsible for structure construction and incremental updates, whereas the data plane design, $\texttt{STEM}^2_\texttt{D}$, is optimized for high-speed lookups. This decoupling is also reflected in the separator design. Specifically, $\texttt{XBSS}_\texttt{C}$, the separator used in $\texttt{STEM}^2_\texttt{C}$, combines a Counting Bloom Filter [11] with an Othello hashing structure [44]. It is then transformed into the data-plane separator $\texttt{XBSS}_\texttt{D}$ by replacing the Counting Bloom Filter with a standard Bloom Filter [3] while retaining the Othello structure. Another key novelty of our design is that we apply less hashing [17] to both XBSS and the overall STEM^2 structure to reduce per-operation computational overhead. In particular, lookups in both $\texttt{XBSS}_\texttt{D}$ and $\texttt{STEM}^2_\texttt{D}$ require only two hash computations, thereby improving query performance.

This paper makes the following contributions.

- We propose STEM^2, a holistic system for exact MS-MQ that adopts a decoupled control-plane/data-plane architecture to jointly achieve 100% query accuracy, high lookup throughput, compact memory usage, and dynamic update support.
- We design XBSS, a novel Exact Binary Set Separator that serves as the core building block of STEM^2. XBSS adopts a decoupled control-plane/data-plane design and incorporates less-hashing technique to support compact, high-throughput exact lookups and efficient updates.
- We propose two splitting strategies for STEM^2, namely fully greedy splitting (FGS) and balanced-then-greedy splitting (BGS), and comprehensively analyze their trade-offs across different scenarios.
- We implement STEM^2 and run comprehensive experiments to show that STEM^2 provides the highest lookup throughput, with the throughput exceeding 120 million queries per second, is memory-efficient, and can support dynamic key updates. Notably, STEM^2 guarantees 100% correctness in query results.

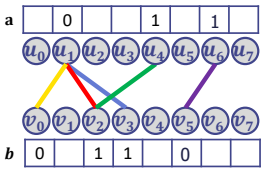
The remainder of this paper is organized as follows. Section 2 reviews related work for multi-set membership queries and binary set separator designs. In Section 3, we present the background and motivation for our study. Section 4 introduces our presented binary set separator XBSS. Section 5 details the design of our proposed system, STEM^2. The evaluation results of STEM^2 are presented in Section 6, and we conclude the paper in Section 7.

2 RELATED WORK

2.1 Multi-set Membership Queries

Conventional MS-MQ solutions map each key to its associated set ID using hash tables, yet explicit key storage often incurs high memory overhead. To optimize performance, the 3D Hash Table [12] employs a hierarchical structure that maintains a single key for multiple values, substantially increasing lookup throughput. Similarly, Cuckoo hashing [28] provides $O(1)$ lookups, with space-efficient variants [9, 21, 38] utilizing compact fingerprints instead of full keys to further minimize memory footprints. More recent work proposes key-value lookup structures that avoid storing full keys altogether, such as SetSep [10, 47], Bloomier/Othello hashing [4, 5, 44], and Ludo hashing [32]. The Bloomier filter was originally designed for static lookup tables and therefore does not support dynamic updates. While Othello hashing [44], an extension of Bloomier filters, enables runtime updates. Among these methods, Ludo hashing achieves the lowest memory cost for dynamic key-value lookups and provides high lookup throughput [32]. Another representative work, Coloring Embedder [36], adopts a Bloomier-like design. Its key idea is to embed each key into a high-dimensional space to minimize hashing collisions, followed by dimensionality reduction to encode set IDs efficiently.

Another line of research explores filter-based data structures for MS-MQ [8, 11, 14, 22, 25, 29, 35, 40, 41, 43]. These methods typically extend Bloom filters [3] or Cuckoo filters [9]. For example, Buffalo [43] builds approximate membership query structures, such as Bloom filters [3] or Cuckoo filters [9], for each set individually, and then checks whether a key belongs to a given set. The Shifting Bloom Filter (ShBF) [41] stores set IDs using offset bits within the filter. However, ShBF does not support dynamic sets or key deletions and suffers from performance degradation as the number of represented sets increases. The Shifting Filter (SF) [14] introduces a modified Cuckoo filter design that supports efficient set queries and key deletions with modest memory usage. The B_h Sequence-based Bloom Filter ($\texttt{B}_\texttt{h}\texttt{BF}$) [29], a variant of the Bloom filter, encodes set IDs as a B_h sequence to compactly represent multi-set memberships and reduce auxiliary storage overhead while supporting insertions and membership queries. The Marked Cuckoo Filter (MCF) [25] extends the Cuckoo filter by attaching identifiers to slots for storing set IDs, enabling MS-MQ with improved space efficiency. While



k	set	$h_a(k)$	$h_b(k)$
	Y(1)	6	5
	N(0)	1	0
	Y(1)	1	2
	Y(1)	1	3
	N(0)	4	2

Figure 1: An Example of 1-bit Othello Hashing.

these filter-based approaches are memory-efficient, they cannot guarantee 100% query accuracy.

2.2 Binary Set Queries

The binary set queries problem determines whether a key $x \in U = P \cup N$ belongs to the positive set P or the negative set N. A straightforward solution is to use a hash table that stores the binary set information as the value for each key. Approximate membership query structures, such as Bloom filters [3], Cuckoo filters [9], and their variants [1, 2, 6, 23, 26, 39], address this problem by encoding keys from one set into the filter. However, this approach introduces false positives and therefore cannot guarantee 100% query accuracy. More recent designs propose memory-efficient structures that achieve exact query results without false positives [18, 34]. The filter cascade [18] constructs a multi-layer Bloom filter system for binary set queries, resolving false positives produced by earlier layers. TinyCR [34] introduces a structure called DASS, which adopts a two-layer design: the first layer is a Cuckoo filter, the second layer is an Othello [44]. Both filter cascade and DASS are memory-efficient, especially when two sets differ significantly in size.

3 PRELIMINARIES AND MOTIVATION

This section first introduces several fundamental data structures that serve as building blocks for both prior state-of-the-art solutions and our proposed design for the MS-MQ problem. We then present a tree-based framework for solving the MS-MQ problem.

3.1 Introduction of Basic Data Structures

3.1.1 Bloom filters and counting Bloom filters. Bloom filters [3] are the most well-known data structures for approximate membership queries. A Bloom filter represents a set of n keys by an array of m bits. Each key x is mapped to k positions in the array by k independent hash functions $h_1, h_2, \ldots, h_l$, and the bits at positions $h_i(x)$ are set to 1 for all $0 \le i \le k-1$. To test whether a key x belongs to the set, the Bloom filter checks the value in the $h_i(x)$-th bit. If all k bits are 1, the Bloom filter returns true. Otherwise, it returns false. A Bloom filter yields false positives. The false positive rate is $\epsilon \approx \left(1 - e^{-kn/m}\right)^k$. For a target false positive rate ϵ, the optimal number of hash functions $k = \frac{m}{n} \ln 2$, which requires an optimal bit array size of $m \approx 1.4427\, n \log_2 \frac{1}{\epsilon}$. One limitation of Bloom filter is that it cannot support key deletions. Counting Bloom filters [11] address this limitation and support deletions by replacing each bit in the array with a counter that records how many times the corresponding position has been set.

3.1.2 Othello. Othello hashing [44] is a key-value lookup structure which can represent a set of key-value pairs. In 1-bit Othello hashing, each value is either 0 or 1, therefore 1-bit Othello is an efficient candidate for membership query problem. Given a set S with n keys, a 1-bit Othello hashing structure is defined as a seven-tuple $< \mathsf{n_a}, \mathsf{n_b}, \mathsf{a}, \mathsf{b}, \mathsf{h_a}, \mathsf{h_b}, \mathsf{G} >$, where:

- n_a and n_b are the sizes of the Othello arrays, with $n_a \approx 1.33n$ and $n_b \approx n$.
- a and b are arrays of n_a and n_b bits, respectively.
- h_a and h_b are uniform random hash functions mapping keys to integers in $\{0, 1, \ldots, n_a - 1\}$ and $\{0, 1, \ldots, n_b - 1\}$, respectively.
- G is an acyclic bipartite graph used to determine the bit values in a and b during Othello construction.

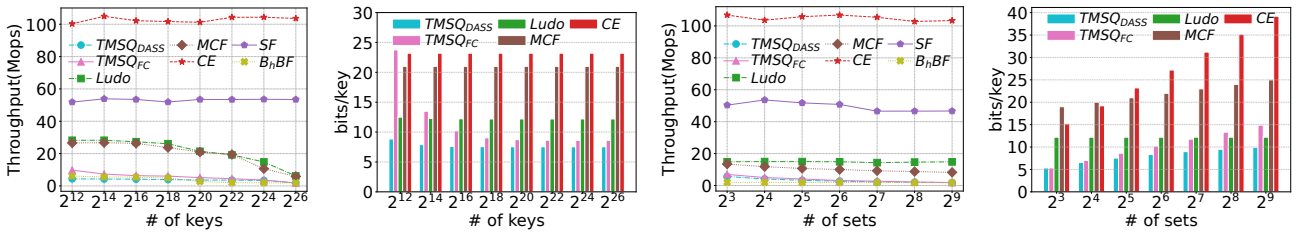
During construction, Othello hashing must form an acyclic bipartite graph when inserting all keys. Once the graph is acyclic, the bit values in a and b are determined accordingly. If two or more keys generate edges that create a cycle, the graph becomes cyclic. In this case, the structure must be reconstructed by selecting a new pair of hash functions until an acyclic graph is obtained. Figure 1 illustrates an example of a 1-bit Othello hashing. The Othello structure is built using five keys, where each key is associated with a set ID of either 0 or 1. To query the value associated with a key x, the lookup result is computed as $\tau(x) = a[h_a(x)] \oplus b[h_b(x)]$.

3.1.3 Binary Set Separators for exact set membership queries. Approximate membership query data structures, built on a finite set P, can answer whether a key belongs to the set. However, they may yield false positives. Given the finite set P and the corresponding negative set N, a binary set separator determines whether a key $x \in U = \{P \cup N\}$ belongs to P or N. Representative examples include Filter Cascade [18] and DASS [34], which provide 100% query accuracy and are more memory-efficient than general-purpose hash tables for key-value storage and lookup.

Filter cascade. The filter cascade introduced in CRLite [18] is a multi-layer Bloom filter structure designed for binary set membership queries. The first-layer filter, BF_1, encodes set P. Due to the false positives inherent to Bloom filters, some keys from set N may be incorrectly classified as members of P. These false positives are collected to construct the second-layer filter, BF_2. Similarly, BF_2 may still yield false positives, which are then used to build the third layer, and so on. This cascading process continues until the false-positive set becomes empty, ensuring 100% query accuracy.

In this data structure, the odd-numbered Bloom filters represent whitelists (encoding keys from P), while the even-numbered filters represent blacklists (encoding keys from N). To determine whether a key x belongs to P or N, the query starts at BF_1 and proceeds layer by layer until the first filter BF_i is found such that $x \notin \mathsf{BF}_\mathsf{i}$. Because Bloom filters never yield false negatives, if i is odd, then x must belong to N; if i is even, then x must belong to P. If no such BF_i is found (i.e., x is contained in all filters), the result depends on the total number of layers l: if l is odd, $x \in P$; otherwise, $x \in N$.

DASS. TinyCR [34] introduces a compact data structure called DASS for binary set membership queries. DASS adopts a two-layer architecture, as illustrated in Figure 3. The first layer is a filter implemented using a Cuckoo filter [9], while the second layer is an Othello structure [44]. Specifically, all keys from set P are first inserted into the Cuckoo filter. Then, keys from set N are tested



(a) Lookup thro. vs. # of keys (b) Memory cost vs. # of keys (c) Lookup thro. vs. # of sets (d) Memory cost vs. # of sets

Figure 2: Comparison of Lookup Throughput and Memory Cost with Varied Number of Keys and Sets.

against the filter. Most keys in N will be correctly identified as negatives, forming the true negative set (TN). However, a small subset of N may be incorrectly identified as positives due to the inherent false positive rate of the filter, forming the false positive set (FP). An Othello is then constructed to distinguish between keys from P (labeled as 1) and keys from FP (labeled as 0). During a query for a key x, if the Cuckoo filter returns a negative result, then $x \in N$, and the query terminates. If the Cuckoo filter returns a positive result, Othello is used to complete the lookup. If Othello returns 1, x is a true positive and thus belongs to P; otherwise, x is a false positive and belongs to N.

Both filter cascade and DASS achieve low memory cost when $|N| \gg |P|$. Compared with filter cascade, DASS additionally supports efficient key updates without requiring reconstruction of the entire data structure.

3.2 Tree-based Framework for Multi-Set Query

Given a binary set separator which can separate keys from two disjoint sets, it is natural to extend it to support MS-MQ by organizing such separators into a hierarchical tree structure, an approach also discussed in TinyCR [34]. Given n sets $\{S_1, S_2, \ldots, S_n\}$, we construct a binary decision tree in which each node employs a binary set separator to partition the keys into two disjoint subsets. The tree growth continues until each leaf node becomes pure, meaning that all keys associated with that leaf belong to a single set S_f. To determine the set ID for a given key x, a binary set query is performed at each node along the path from the root to a leaf. At each step, the query result decides the traversal direction, left or right, until a leaf node is reached.

Despite this intuitive design, existing tree-based solutions for MS-MQ suffer from suboptimal lookup throughput and memory overhead [20]. Although incorporating DASS into the tree-based framework for MS-MQ on skewed datasets reduces memory cost [34], it still suffers from low lookup throughput, as shown in Figure 2.

Preliminary evaluation. We incorporate DASS and filter cascade into the tree-based framework for MS-MQ, denoted as $\texttt{TMSQ}_{\texttt{DASS}}$ and $\texttt{TMSQ}_{\texttt{FC}}$, respectively, and evaluate their lookup throughput and memory cost. To exploit the property that both DASS and filter cascade achieve lower memory cost when $|N| \gg |P|$, we adopt a greedy strategy to determine the splitting point at each tree node. Specifically, to construct a binary set separator at a node for a given collection of sets $\{S_1, S_2, \ldots, S_n\}$, we first sort by their sizes and select the median set as the splitting point. This approach maximizes the size ratio between the left and right child

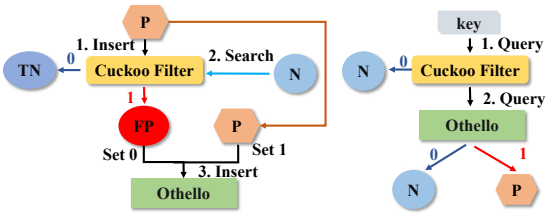


Figure 3: Insertion and Query Process of DASS.

nodes, resulting in a balanced tree with minimal depth. We generate synthetic datasets with Zipfian-distributed set sizes to evaluate their performance against existing state-of-the-art methods, including Ludo Hashing [32], Shifting Filter (SF) [14], B_h Sequence-based Bloom Filter ($\texttt{B}_\texttt{h}\texttt{BF}$) [29], Marked Cuckoo Filter (MCF) [25], and Coloring Embedder (CE) [36] (see Figure 2). Specifically, we fix the number of sets to 2^5 in Figure 2(a) and Figure 2(b), and fix the total number of keys to 2^{24} in Figure 2(c) and Figure 2(d). The results show that both $\texttt{TMSQ}_{\texttt{FC}}$ and $\texttt{TMSQ}_{\texttt{DASS}}$ achieve high memory efficiency in most scenarios, with $\texttt{TMSQ}_{\texttt{DASS}}$ even outperforming other baselines. However, both approaches exhibit lower lookup throughput compared to other methods.

3.3 Analysis of Limitations and Design Insights

The low lookup throughput in the tree-based framework is mainly due to the traversal of multiple binary set separators, each of which requires several hash computations and memory accesses for a single key lookup. Moreover, $\texttt{TMSQ}_{\texttt{FC}}$ does not support key deletions because of the inherent limitations of Bloom filters, and key insertions can incur significant overhead, often necessitating reconstruction of the filter cascade. Despite these limitations, the tree-based structure remains a promising approach for achieving 100% query accuracy with low memory usage. The key challenge, therefore, is to design a tree-based structure that delivers high lookup throughput while supporting efficient updates, including insertions, deletions, and key migrations.

New design insights. The design of MS-MQ data structures requires high lookup throughput, 100% query accuracy, low memory cost, and support for dynamic updates. The experimental results in Figure 2 show that the tree-based structures $\texttt{TMSQ}_{\texttt{FC}}$ and $\texttt{TMSQ}_{\texttt{DASS}}$ achieve low memory cost, but both suffer from low lookup throughput. To understand this limitation, we analyze the design of Bloom filters and Cuckoo filters. Bloom filters have simpler memory access patterns than Cuckoo filters and their variants, but their lookup

efficiency is often limited by the computational overhead of multiple hash functions. This observation suggests that, if Bloom filters can be effectively adapted to construct binary set separators while minimizing their hashing overhead within a tree-based framework, the resulting structure could support highly efficient MS-MQ.

Furthermore, practical MS-MQ deployments often operate over a heterogeneous memory hierarchy. While the complete and dynamically evolving collection of sets resides in large-capacity memory, the MS-MQ structure itself must fit within a much smaller fast-memory footprint. This separation may arise across devices—for example, when a centralized server maintains the full set collection while a compact query structure is offloaded to resource-constrained devices such as routers or IoT nodes. It can also arise within a single device, where structure construction and updates are handled in larger but slower memory, while lookup queries are served from smaller and faster memory. This resource asymmetry is particularly well aligned with MS-MQ workloads, where query processing must be fast and memory-efficient, but construction and updates can be handled separately. It therefore motivates a decoupled design with two distinct planes: a control plane for constructing the data structure and processing incremental updates, and a data plane optimized exclusively for high-speed queries.

Guided by these dual-plane insights, the following sections first detail the **Exact Binary Set Separator (**XBSS**)** as a dual-plane building block, followed by the comprehensive design for MS-MQ.

4 DESIGN OF THE LIGHTWEIGHT XBSS

In this section, we present XBSS, a lightweight binary set separator designed for exact set membership queries, which serves as the core building block of our MS-MQ design.

Adhering to the architectural principles in Section 3.3, XBSS adopts a decoupled design to reconcile query efficiency with update agility, as shown in Figure 4. Structure construction and update maintenance are handled in the control plane ($\texttt{XBSS}_\texttt{C}$), while the data plane ($\texttt{XBSS}_\texttt{D}$), is optimized for high-throughput lookups and memory compactness. This separation isolates maintenance overhead from the performance-critical query path and enables seamless transformation into a query-optimized representation.

4.1 Design of $\texttt{XBSS}_\texttt{C}$

$\texttt{XBSS}_\texttt{C}$ adopts a two-layer structure inspired by DASS [34], as shown in Figure 4(a). The first layer is a counting Bloom filter (CBF) [11] which supports dynamic updates, including key insertion and deletion. The second layer is an Othello. However, directly incorporating the CBF into the design leads to substantial computational overhead, as it requires multiple hash computations for each operation. Inspired by the findings that two independent hash functions $h_1(x)$ and $h_2(x)$ can simulate multiple hash functions by Equation 1:

$$g_i(x) = h_1(x) + ih_2(x) \qquad (i = 0, 1, 2, ...) \tag{1}$$

without any loss in the asymptotic false positive probability [17], this **less hashing technique** provides a promising solution to reduce the computational overhead. Accordingly, we adopt this technique in $\texttt{XBSS}_\texttt{C}$, reducing the total number of hash computations required for key operations (e.g., insertion, flipping, and deletion) to two, thereby enabling a lightweight counting Bloom filter. Thus, $\texttt{XBSS}_\texttt{C}$ selects two base hash functions to derive the hash functions required by the CBF and Othello. Since Othello may trigger a reconstruction process when the selected hash functions fail to produce an acyclic graph, the two base hash functions may need to be reselected. $\texttt{XBSS}_\texttt{C}$ supports the following operations:

Construction ($\texttt{XBSS}_\texttt{C}$.build**).** Given two disjoint groups of keys, $\texttt{D}_\texttt{L}$ and $\texttt{D}_\texttt{S}$, the control plane constructs $\texttt{XBSS}_\texttt{C}$ to distinguish between them. A CBF is first built using the keys from $\texttt{D}_\texttt{S}$. Then, the keys from $\texttt{D}_\texttt{L}$ are queried against this filter, where most are expected to yield negative results. Keys that return positive results constitute the false positive set FP (set 0). An Othello structure is then built to distinguish between FP (set 0) and the true positive set $\texttt{D}_\texttt{S}$ (set 1).

New key insertion ($\texttt{XBSS}_\texttt{C}$.insert(x)**).** For a new key x inserted into set $\texttt{D}_\texttt{S}$, the key should be added to both the CBF and the Othello O. If $\texttt{CBF.query}(x) == 1$, then x only needs to be inserted into O with $O.\texttt{query}(x) == 1$. Otherwise, x is first inserted into the CBF. After the insertion, keys from $\texttt{D}_\texttt{L}$ are rechecked against the updated CBF to identify any new false positives. These newly identified keys x' are inserted into O with $O.\texttt{query}(x') == 0$, while the newly inserted key x from $\texttt{D}_\texttt{S}$ is inserted into O with $O.\texttt{query}(x) == 1$. For a new key x inserted into set $\texttt{D}_\texttt{L}$, if $\texttt{CBF.query}(x) == 0$, no further action is required. Otherwise, x should be inserted into O with $O.\texttt{query}(x) == 0$.

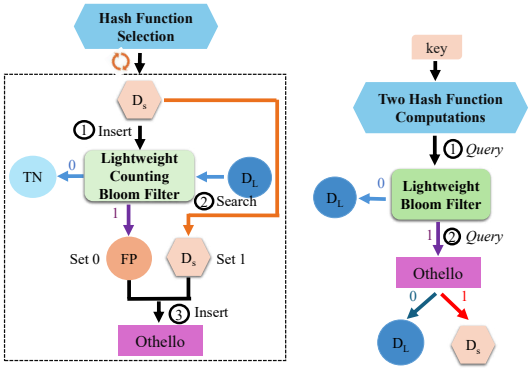
Key flipping ($\texttt{XBSS}_\texttt{C}$.flip(x)**).** When a key x is moved from $\texttt{D}_\texttt{S}$ to $\texttt{D}_\texttt{L}$, CBF first removes x since x has already been inserted into CBF and O, and then checks whether x would be recognized as a false positive key after removal. If x is a false positive, then Othello O flips the value of x by making $O.\texttt{query}(x) == 0$. Otherwise, x is deleted from O. When a key x is moved from $\texttt{D}_\texttt{L}$ to $\texttt{D}_\texttt{S}$, CBF first checks whether x is tested positive. If so, Othello O flips the value of x by making $O.\texttt{query}(x) == 1$. Otherwise, x is inserted into CBF and $\texttt{D}_\texttt{L}$ are tested against CBF to get new false positives. Then x is inserted into O with $O.\texttt{query}(x) == 1$ and new false positives x' is inserted into O with $O.\texttt{query}(x') == 0$.

Key deletion ($\texttt{XBSS}_\texttt{C}$.delete(x)**).** When a key x is removed from $\texttt{D}_\texttt{S}$, then both CBF and O delete x. When a key x is removed from $\texttt{D}_\texttt{L}$, CBF first checks whether x is a false positive. If so, O deletes x. Otherwise, no further operation is required.

It is worth noting that insertions or deletions in CBF may alter the set of false positives when testing keys from $\texttt{D}_\texttt{L}$. Therefore, these keys need to be re-evaluated. To reduce the overhead of re-checking all keys in $\texttt{D}_\texttt{L}$, we maintain an index table associated with the CBF. The index table records, for each slot in the CBF, the indices of the $\texttt{D}_\texttt{L}$ keys that hash to that slot. When the value of a slot changes due to an insertion, flipping, or deletion, only the keys linked to that slot need to be re-checked, and the false positive set can then be updated accordingly.

4.2 Design of $\texttt{XBSS}_\texttt{D}$

After $\texttt{XBSS}_\texttt{C}$ has been successfully constructed using the selected two base hash functions, $\texttt{XBSS}_\texttt{D}$ is obtained by converting the CBF into a standard Bloom filter (BF), setting all nonzero counters to 1, and copying the Othello structure, as shown in Figure 4(b). The resulting $\texttt{XBSS}_\texttt{D}$ can then be deployed in the data plane to support efficient exact binary set membership queries. The main supported operation of $\texttt{XBSS}_\texttt{D}$ is key lookup.



(a) $XBSS_C$ for control plane construction and update (b) $XBSS_D$ for data plane lookup

Figure 4: Design of XBSS on Control Plane and Data Plane.

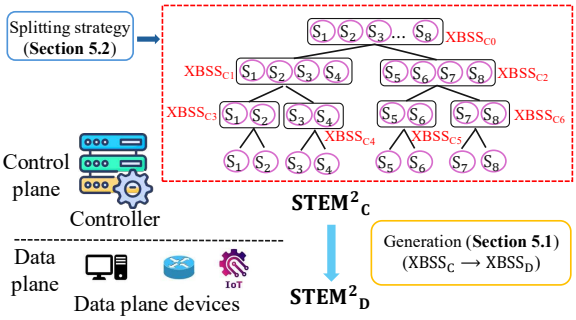


Figure 5: System Overview of STEM^2 for Supporting MS-MQ.

Key lookup (XBSS$_D$.lookup(x)). For a given key x, only two hash computations are required for the query. if BF.query(x) == 0, then $x \in D_L$. Otherwise, the key is checked in the Othello O. If O.query(x) == 0, then $x \in D_L$. Otherwise, $x \in D_S$.

4.3 Memory Analysis of XBSS

This section analyzes the memory cost of XBSS, focusing on the trade-offs under different false positive rate configurations of the filter. We examine $XBSS_D$, which is deployed in the data plane and may operate on resource-constrained devices. $XBSS_D$ shares the same structural properties as $XBSS_C$, except that the CBF in $XBSS_C$ requires approximately c times more memory than the BF used in $XBSS_D$, where c denotes the number of bits per counter in the CBF.

$XBSS_D$ consists of a Bloom filter BF and an Othello O. A key design trade-off exists between the sizes of BF and O, determined by the false positive rate of BF. Specifically, reducing the false positive rate of BF requires more memory but results in fewer false positives, thereby reducing the size of O needed to distinguish between false positives and true positives. Suppose $XBSS_D$ is built using keys from two groups D_S and D_L with size n_s and n_l respectively ($n_l \geq n_s$), and keys from D_S are inserted into the BF. If the false positive rate of BF is set to ϵ, then BF requires at least $1.4427 n_s \cdot \log_2(\frac{1}{\epsilon})$ bits [3]. The expected number of false positives is $\epsilon \cdot n_l$. The Othello O then requires $2.33 \cdot (\epsilon \cdot n_l + n_s)$ bits [44]. Thus the total memory cost of XBSS is $1.4427 \cdot n_s \cdot log_2(\frac{1}{\epsilon}) + 2.33 \cdot n_l \cdot \epsilon + 2.33 \cdot n_s$.

Let $r = n_l/n_s$, then the total memory cost is $M = n_s \cdot (1.4427 \cdot log_2(\frac{1}{\epsilon}) + 2.33 \cdot r \cdot \epsilon + 2.33)$. Since n_s and r are constant for the given sets, we can minimize M with respect to ϵ. The minimum memory cost is achieved when $\epsilon = \frac{0.8925}{r}$, yielding $M_{min} = (2.08 \ln r + 4.65) n_s$. Accordingly, the amortized memory cost per key is $M_a = (2.08 \ln r + 4.65)/(r+1)$. By taking the derivative of M_a with respect to r and setting $\frac{\partial M_a}{\partial r} = 0$, we find that M_a reaches its maximum at $r = 0.892$. Since $r \geq 1$ in practical settings, increasing r monotonically reduces the amortized memory cost per key.

5 MULTI-SET MEMBERSHIP QUERY DESIGN

This section presents the holistic design of STEM^2. As shown in Figure 5, STEM^2 adopts a decoupled architecture composed of a control plane and a data plane. Following the principles established in Section 3.3, maintenance-intensive operations, including structure construction and incremental updates, are assigned to the control plane $STEM^2_C$. In contrast, the data plane $STEM^2_D$ remains a lightweight, read-only structure for high-throughput, low-latency lookups on data-plane devices. We then present the splitting strategies used to construct $STEM^2_C$ in Section 5.2.

5.1 Design of $STEM^2_C$ and $STEM^2_D$

We adopt the tree-based framework to construct $STEM^2_C$. At each node of the tree, we incorporate our proposed $XBSS_C$, introduced in Section 4, as the binary set separator. The corresponding data-plane structure, $STEM^2_D$, is obtained by replacing each $XBSS_C$ in $STEM^2_C$ with its data-plane counterpart, $XBSS_D$. To enable a computationally efficient design, we also adopt the less-hashing technique [17]. Specifically, two base hash functions are selected to simulate all hash functions used in $STEM^2_C$ and $STEM^2_D$, thereby reducing the total number of hash computations for key lookups on $STEM^2_D$ to two.

5.2 Strategy for Choosing a Splitting Point

With the efficient binary separator XBSS, we construct a tree-based structure, STEM^2, to support MS-MQ. However, the choice of how multiple sets are partitioned at each tree node can lead to trees with substantially different shapes and depths, which in turn affects the lookup throughput, memory overhead, and update efficiency of STEM^2. To jointly optimize these metrics, it is crucial to carefully select the splitting point used to partition the sets at each tree node.

We first analyze that splitting among a set is not optimal. XBSS is built to separate keys to two partitions, L and R. As discussed in Section 4.3, The memory efficiency of XBSS improves as the ratio $r = |R|/|L|$ increases, where $|L|$ and $|R|$ are the sizes of sets L and R, respectively. However, when r is small (especially when $|R| \approx |L|$), the memory cost of the XBSS ($M_o = (2.08 \ln \frac{|R|}{|L|} + 4.65)\,|L|$) may become prohibitively high. An intuitive approach to mitigating this is to introduce a threshold θ, and adjust the splitting point to make $r = \theta$, by allowing a set to be split across two partitions. Taking this strategy, we will first divide L into two sets L_1 and L_2 to make $r = (|R| + |L_2|)/|L_1| = \theta$. Then for sets R and L_2, we may take the same step until two sets L_n and R meet the ratio requirement. Thus, during this process, the original two sets are divided into multiple sets $L_1, L_2, ..., L_n, R$ and we need to build $n-1$ XBSS instances to finally separate L and R. The total memory cost for this approach will be $M_i = 2.08 \sum_{i=1}^{n-1} (\ln \frac{|L_{i+1}|+...+|L_n|+|R|}{|L_i|} +$

4.65) $|L_i|+(2.08\ln\frac{|R|}{|L_n|}+4.65)\,|L_n|$. Given $|L_1|+|L_2|+|L_3|+\ldots+|L_n| = |L|$, $M_i - M_o = 2.08\sum_{i=1}^{n-1}\ln\frac{|L_{i+1}|+\ldots+|L_n|+|R|}{|L_i|}\,|L_i| + 2.08\ln\frac{|R|}{|L_n|}\,|L_n| - 2.08\sum_{i=1}^{n}\ln\frac{|R|}{|L|}\,|L_i| > 0$. This shows that splitting a single set into multiple partitions to obtain a higher r does not reduce memory cost; instead, it introduces more XBSS instances and consequently increases lookup latency. Therefore, when designing the splitting strategy, we do not place splitting points within a set.

We then propose two splitting strategies: fully greedy splitting (FGS) strategy and balanced-then-greedy splitting (BGS) strategy.

5.2.1 FGS strategy. As discussed in Section 4.3, for a XBSS, selecting a splitting point that maximizes the size ratio $r = \mathsf{n_l}/\mathsf{n_s}$ between the two groups $\mathsf{D_s}$ and $\mathsf{D_l}$ minimizes the memory overhead of $\mathtt{XBSS}_{\mathsf{D}}$. Accordingly, applying a greedy strategy that minimizes the memory cost of each XBSS at tree node—referred to as fully greedy splitting (FGS)—yields the minimum memory overhead of $\mathtt{STEM}^2_{\mathsf{D}}$. Formally, consider n sets $S_0, S_1, \ldots, S_{n-1}$ sorted in nondecreasing order of size, i.e., $|S_0| \le |S_1| \le \ldots \le |S_{n-1}|$, with total size $T = \sum_{i=0}^{n-1}|S_i|$. Under the FGS strategy, the k-th split ($0 \le k \le n-2$) partitions the current collection $S_k, S_{k+1}, \ldots, S_{n-1}$ into a left group S_k and a right group $S_{k+1}, \ldots, S_{n-1}$. The total memory cost of the resulting tree is $M_1 = \sum_{k=0}^{n-2}\big(2.08|S_k|\ln\frac{R_k}{|S_k|} + 4.65|S_k|\big)$, where $R_k = \sum_{i=k+1}^{n-1}|S_i|$.

5.2.2 BGS strategy. To reduce tree depth and improve lookup throughput as well as dynamic update efficiency, we first construct a balanced tree and then apply a greedy strategy at each node to minimize the local memory cost, following the approach described in Section 3.2. This method, referred to as balanced-then-greedy splitting (BGS), reduces the tree depth to $\lceil\log_2(n)\rceil$ for n sets but increases the data plane memory overhead. Under the BGS strategy, the data-plane memory cost of $\mathtt{STEM}^2_{\mathsf{D}}$ is $M_2 = \sum_{nodes}(2.08\ln(n_r/n_l)+4.65)n_l$ where n_r and n_l denote the number of keys in the two subgroups at each node.

5.2.3 Analysis. Because the overall memory cost of $\mathtt{STEM}^2_{\mathsf{D}}$ is dominated by the linear merging component, the efficiency of each strategy depends on how often a set of size $|S_k|$ is accumulated into the left partition. For M_1, the linear cost is $C_{lin}(M_1) = 4.65\sum_{k=0}^{n-2}|S_k|$, which ensures that each set—except the largest one, $|S_{n-1}|$—is counted exactly once in the memory accumulation. In contrast, BGS organizes keys as a balanced hierarchical tree, where smaller sets are repeatedly assigned to the left subtree. Let $z(k)$ denote the number of times $|S_k|$ is placed in the left partition across the hierarchy. Then, the linear cost of M_2 can be expressed as $C_{lin}(M_2) = 4.65\sum_{k=0}^{n-1}z(k)\,|S_k|$. Since $z(k) \ge 1$ for all $k < n-1$, and can be as large as $\log_2 n$ for the smallest sets, these set sizes are repeatedly accumulated across multiple tree levels. Consequently, M_1 achieves a strictly smaller memory footprint than M_2, i.e., $C_{lin}(M_1) < C_{lin}(M_2)$.

However, from the perspectives of lookup throughput and dynamic update efficiency, applying the FGS strategy at each node produces a highly unbalanced tree with depth n. Consequently, querying or updating keys belonging to larger sets requires traversing more tree levels, which degrades both lookup throughput and update efficiency. For example, a query for a key belonging to S_{n-1} must pass through all $n-1$ XBSS nodes, significantly increasing lookup latency and reducing throughput. Moreover, the FGS strategy increases the control-plane memory overhead of $\mathtt{XBSS}_{\mathsf{C}}$, which in turn raises the overall memory cost of $\mathtt{STEM}^2_{\mathsf{C}}$. The main reason is that a larger ratio $r = \mathsf{n_l}/\mathsf{n_s}$ results in a larger index table in the CBF, because the indices of all keys in $\mathsf{D_l}$ must be maintained in that table. In contrast, the BGS strategy generates more balanced partitions at each node, resulting in a smaller size ratio $r = \mathsf{n_l}/\mathsf{n_s}$ when constructing XBSS nodes. This smaller ratio reduces the size of the index table in the control-plane CBF of $\mathtt{XBSS}_{\mathsf{C}}$, and thus lowers the overall memory overhead of $\mathtt{STEM}^2_{\mathsf{C}}$.

In summary, users can select between the two splitting strategies according to their specific requirements, further enhancing the flexibility of STEM^2 deployments. Unless otherwise specified, we adopt the BGS strategy in the implementation of STEM^2 to achieve higher lookup throughput and more efficient updates.

5.3 Operations of STEM^2

5.3.1 Operations of $\mathtt{STEM}^2_{\mathsf{C}}$. The control plane is responsible for constructing $\mathtt{STEM}^2_{\mathsf{C}}$ and updating it in response to key updates.

Construction. Given multiple sets $\{S_1, S_2, \ldots, S_n\}$, we first select the two base hash functions and use them to construct a balanced tree, applying the splitting strategy at each node to determine the splitting point. This process partitions the sets into left and right groups, which are then used to build a binary set separator XBSS at each node. $\mathtt{STEM}^2_{\mathsf{C}}$ invokes the $\mathtt{XBSS}_{\mathsf{C}}$.build function to construct the binary separator. Specifically, the hash functions used in each $\mathtt{XBSS}_{\mathsf{C}}$ are derived from the two base functions by setting different indices i, as defined in Equation 1. Notably, if an Othello of XBSS fails to find simulated hash functions to build an acyclic graph after exceeding a predefined number of attempts, then $\mathtt{STEM}^2_{\mathsf{C}}$ will reselect two base functions and retry the construction.

Insertion and deletion. Inserting or deleting a key x from a set triggers updates to all XBSS instances along the path from the target leaf node to the root. At each node, the corresponding XBSS invokes $\mathtt{XBSS}_{\mathsf{C}}$.insert(x) to add the key or $\mathtt{XBSS}_{\mathsf{C}}$.delete(x) to remove it.

Inter-set key migration. A key migration refers to changing the set ID of a key. Suppose a key x is migrated from set i to set j; this operation is equivalent to inserting x into set j and deleting x from set i. Thus, the migration is performed by executing the corresponding insertion and deletion operations.

Batch update. A batch update mechanism is introduced to reduce amortized update latency. Each $\mathtt{XBSS}_{\mathsf{C}}$ in STEM^2 caches the SetID in its two partitions. For each key update, STEM^2 first uses the cache information to identify the affected $\mathtt{XBSS}_{\mathsf{C}}$ instances. Then, STEM^2 derives the corresponding operations—i.e., insertion, deletion, flipping—for each affected instance. For a key insertion or deletion, the corresponding update is the insertion or deletion operations on all affected $\mathtt{XBSS}_{\mathsf{C}}$ instances. A key migration is decomposed into deleting the key from its original set and inserting it into the target set, yielding two ordered lists of separator-level operations. These two lists are then merged from bottom to top until the first common $\mathtt{XBSS}_{\mathsf{C}}$ is encountered, at which point the insertion and deletion are combined into a flipping operation. For example, if a key x is migrated from S_1 to S_4 in Figure 5, STEM^2 generates the operations $\mathtt{XBSS}_{\mathsf{C3}}$.delete(x), $\mathtt{XBSS}_{\mathsf{C4}}$.insert(x), and $\mathtt{XBSS}_{\mathsf{C1}}$.flip(x). After all key updates have been analyzed, the generated operations are grouped by each $\mathtt{XBSS}_{\mathsf{C}}$ and executed in batches.

5.3.2 Operations of $\texttt{STEM}^2_\texttt{D}$. **Lookup.** To determine the set ID of a given key x, the query begins from the $\texttt{XBSS}_\texttt{D}$ instance at the root node. The result of $\texttt{XBSS}_\texttt{D}.\texttt{lookup}(x)$ indicates whether x belongs to the left or right partition, thereby determining the next $\texttt{XBSS}_\texttt{D}$ instance to query. This process continues recursively until a leaf node is reached, at which point the set ID associated with that leaf is returned as the final result.

6 EVALUATION

6.1 Implementation and Experiment Setup

We implement a complete software prototype of STEM^2 in C++. The false positive rate ϵ of the (counting) Bloom filter in each XBSS is the only parameter that needs to be configured. We set $\epsilon = \frac{0.8925}{r}$ to minimize memory cost, where r ($r > 1$) denotes the size ratio between the two partition groups. Each counter in the CBF is allocated 4 bits. Once ϵ is determined, the remaining parameters of the CBF and Othello within XBSS are automatically derived, as analyzed in Section 4.3. We conduct two types of performance evaluation. 1) Synthetic-dataset evaluation, where the sizes of different sets follow Zipfian, normal, and uniform distributions. 2) Case study of STEM^2 on two applications, including packet forwarding and transaction tag query. All experiments are run on an Exxact Valence full-tower workstation with AMD Ryzen Threadripper PRO 5965WX, 3.8GHz, 128 MB L3 cache and Ubuntu 22.04.

Metrics. We employ the following metrics to evaluate STEM^2. **1) Lookup throughput**: quantified by the number of lookups a data structure on the data plane can execute per second, reported in million operations per second (Mops). **2) Memory cost**: amortized number of bits consumed per key on the data plane. **3) Update efficiency**: amortized latency to insert/delete/migrate a key on the control plane. It is worth noting that we do not evaluate **lookup accuracy**, as STEM^2 guarantees 100% correctness.

Baseline methods. We compare STEM^2 with existing state-of-the-art data structures and algorithms for MS-MQ, including Ludo hashing (Ludo) [32], coloring embedder (CE) [36], Shifting Filter (SF) [14], B_h sequence-based Bloom filter ($\texttt{B}_\texttt{h}\texttt{BF}$) [29], and Marked Cuckoo filter (MCF) [25]. We use the publicly available C++ implementations of Ludo [31] and CE [37]. For Shifting Filter, we rewrite its public Python implementation [13] in C++. We receive the implementation of $\texttt{B}_\texttt{h}\texttt{BF}$ in Java from its authors and then rewrite it in C++. Finally, we implement MCF in C++ based on the design described in its paper [25]. Meanwhile, we also compare STEM^2 with $\texttt{TMSQ}_\texttt{FC}$ and $\texttt{TMSQ}_\texttt{DASS}$ introduced in Section 3.2. Among those algorithms, only Ludo, $\texttt{TMSQ}_\texttt{FC}$ and $\texttt{TMSQ}_\texttt{DASS}$ can achieve 100% lookup accuracy. To ensure a fair comparison and eliminate the influence of hash function variability on algorithm performance, we use Google FarmHash [16] as the hash function for all evaluated algorithms.

Datasets. We generate synthetic datasets with varying numbers of sets and total keys, where the set size distributions follow Zipfian, normal, and uniform distributions, respectively. Each key is 64 bits.

For two case studies, we employ two real world datasets, and Table 1 summarizes the statistical information of the real datasets.

Network traffic [15]. We utilize a network traffic dataset from Internet of Things (IoT) devices for packet forwarding, where the packet ID is the key and the port number serves as the set ID.

Table 1: Statistical Information of Real Datasets.

Dataset	#Sets	#Keys	min set size	max set size
Network	302	570397	16	271561
Transactions	503	1048575	1	141886

Credit card transactions [19]. This dataset stores the transaction history of users that includes the amount for each transaction. We divide the transactions into different sets according to the transaction amount. For example, the transaction amount between 0 and 5 are assigned to set 0, amounts between 6 and 10 to set 1, and so on. Transactions with amounts greater than 2500 are grouped into one set. Notably, empty sets are discarded.

6.2 Data Plane Evaluation of STEM^2

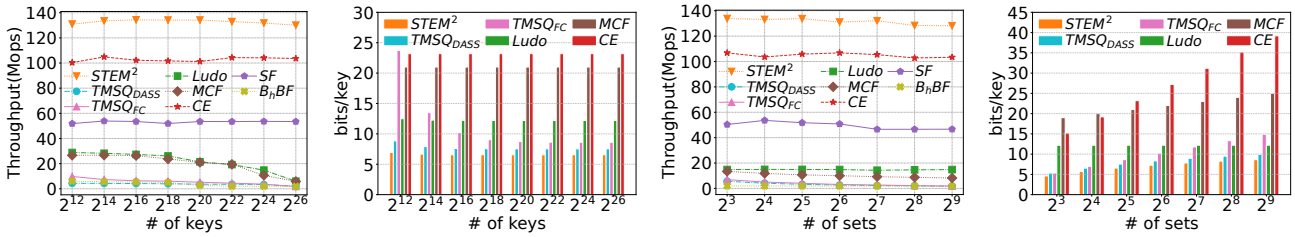
In this section, we evaluate the lookup throughput and memory cost of STEM^2 using synthetic datasets on the data plane.

6.2.1 Performance on Zipfian-distributed dataset. We generate synthetic datasets with varying number of sets and total keys, where the set sizes for each dataset follow a Zipfian distribution.

Lookup throughput and memory cost vs. number of keys. Figure 6(a) and Figure 6(b) present the lookup throughput and memory cost as the total number of keys increases from 2^{12} to 2^{26}, with the number of sets fixed at 2^5. The results show that **STEM^2 achieves exceptionally high lookup throughput**, exceeding 130 Mops, which is more than 25.6%–32.5% higher than the best baseline CE. Moreover, STEM^2 achieves 4.54×–21.63× higher lookup throughput compared to Ludo, which also guarantees fully accurate query. The lookup throughput of other methods ranges from 2 to 55 Mops. As number of keys grows, STEM^2 experiences a negligible degradation in throughput, due to an increased number of cache misses caused by larger data structures when handling more keys. Ludo and MCF show noticeable performance degradation as key size grows.

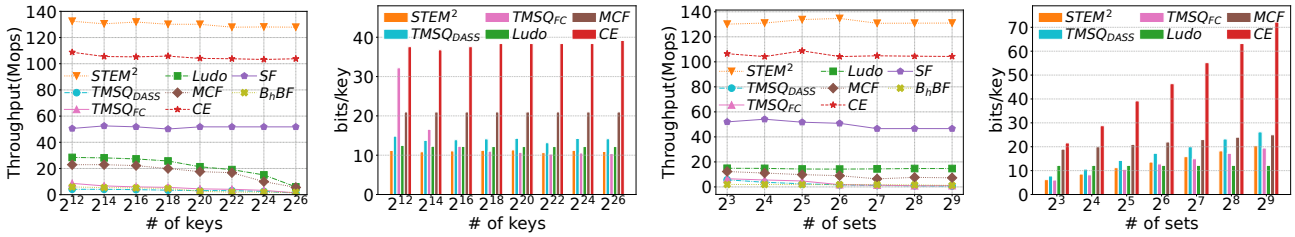
Figure 6(b) shows the memory cost of STEM^2 on the data plane as the number of keys varies. The results show that **STEM^2 is the most space-efficient data structure under Zipfian distribution compared to other methods.** Specifically, STEM^2 consumes 6.56–6.95 bits per key with varying number of keys. $\texttt{TMSQ}_\texttt{DASS}$, has the same tree-based framework as STEM^2 but uses different binary separators, requires 7.54–8.85 bits per key, demonstrating the memory efficiency of the tree-based framework. The memory cost of CE is relatively large compared with STEM^2, which requires around 23.2 bits per key, although it achieves the best lookup throughput among baselines. We do not show memory cost of $\texttt{B}_\texttt{h}\texttt{BF}$ and SF in the figures, as both require significantly more memory space compared to other methods. Specifically, $\texttt{B}_\texttt{h}\texttt{BF}$ is reported to require an average of 80 bits per key based on the bit-field compression, which incurs additional costs in the form of memory alignment overhead and bitwise operation overhead. Allocating a full field such as uint32_t for each counter significantly worsens memory cost (nearly 422 bits per key) but also significantly increases lookup throughput. Our evaluation for throughput and memory cost of $\texttt{B}_\texttt{h}\texttt{BF}$ is based on the full field counter version. SF requires an average of 94.8 bits per key to achieve 95% query accuracy.

Lookup throughput and memory cost vs. number of sets. Figure 6(c) and Figure 6(d) show the lookup throughput and memory



(a) Lookup thro. vs. # of keys (b) Memory cost vs. # of keys (c) Lookup thro. vs. # of sets (d) Memory cost vs. # of Sets

Figure 6: Lookup Throughput and Memory Cost Using Datasets under Zipfian Distribution.



(a) Lookup thro. vs. # of keys (b) Memory cost vs. # of keys (c) Lookup thro. vs. # of sets (d) Memory cost vs. # of Sets

Figure 7: Lookup Throughput and Memory Cost Using Datasets under Normal Distribution.

cost as the number of sets increases from 2^3 to 2^9, with the number of keys fixed at 2^{24}. We observe that the number of sets has a negligible effect on the lookup throughput, and **STEM^2 consistently achieves the highest performance**. Specifically, STEM^2 achieves 22.7%–28.5% higher lookup throughput than CE, and outperforms Ludo by 8.65×–9.23×. As the number of sets increases, a query must traverse more XBSS instances, however, the throughput shows only slight fluctuations. The key reason is that STEM^2 requires only two hash computations in a query process with the less-hashing technique, regardless of the number of XBSS instances in the tree.

In terms of memory cost, **STEM^2 consumes the least memory** among all evaluated methods on the Zipfian-distributed dataset as shown in Figure 6(d). As the number of sets increases, STEM^2's memory usage rises moderately from 4.62 to 8.63 bits per key, primarily due to the introduction of additional XBSS instances for more sets. Among the baselines, TMSQ$_{\text{DASS}}$, the most memory-efficient baseline, requires around 5.33–9.91 bits per key. CE, which achieves the highest lookup throughput among baselines, incurs significantly higher memory cost, ranging from 15.2–39.2 bits per key.

6.2.2 Performance on normally distributed dataset. In some real-world scenarios, the set sizes exhibit a normal-like distribution. To evaluate STEM^2 under such conditions, we construct synthetic datasets with different numbers of sets and total keys, where the set sizes for each dataset follow a normal distribution.

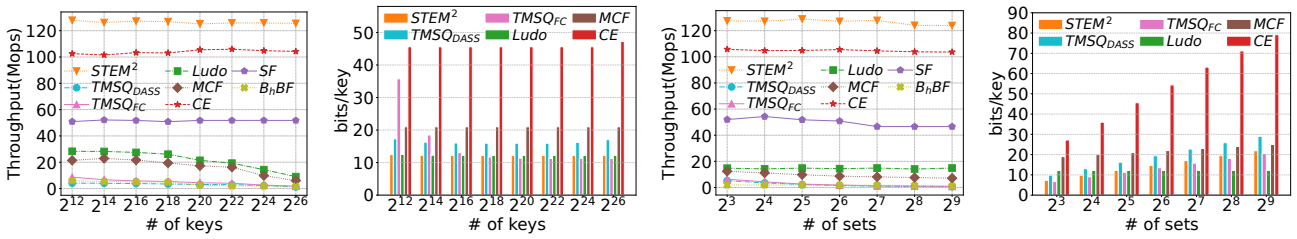
Lookup throughput and memory cost vs. number of keys. Figure 7(a) and Figure 7(b) show the lookup throughput and memory cost of different methods as the total number of keys grows from 2^{12} to 2^{26}, while keeping the number of sets fixed at 2^5. **STEM^2 achieves the highest lookup throughput**, which is 21.8%–25.3% higher than the best baseline method CE. All methods exhibit lookup throughput trends that are consistent with those observed under Zipfian-distributed datasets as the number of keys increases.

For memory cost, **STEM^2 remains the most efficient solution**, with memory usage ranging from 10.65 to 11.34 bits per key. Among baselines, the most efficient design, Ludo, requires an average of 12.2 bits per key on the data plane. Although the normally distributed dataset is less skewed than the Zipfian-distributed one, STEM^2 presents only negligible performance degradation, demonstrating the adaptability and robustness of our design.

Lookup throughput and memory cost vs. number of sets. Figure 7(c) and Figure 7(d) show the lookup throughput and memory cost as the number of sets grows from 2^3 to 2^9, while keeping the total number of keys at 2^{24}. **STEM^2 achieves the highest lookup throughput** among all methods and the increasing number of sets has negligible impact on its throughput.

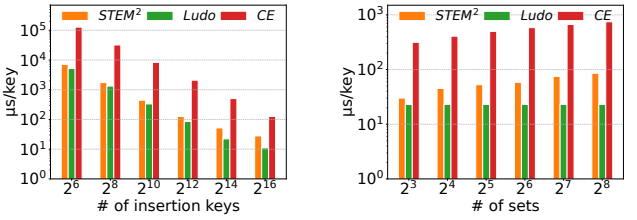
STEM^2 consumes 6.22–20.46 bits memory per key as the number of sets grows, mainly because more XBSS instances are required. Since the normal distribution is less skewed than the Zipfian distribution, STEM^2 experiences some degradation in memory efficiency under this setting. In contrast, Ludo exhibits the lowest and most stable memory usage, requiring 12.2 bits per key regardless of the number of sets. TMSQ$_{\text{FC}}$, as the second-best benchmark in this scenario, requires 6.06–19.47 bits per key. Although STEM^2 does not achieve the lowest memory cost when handling a large number of sets, its memory usage remains comparable to the most efficient methods (Ludo and TMSQ$_{\text{FC}}$) and still outperforms other baselines. Notably, the **superior lookup throughput of STEM^2** makes it a compelling and practical solution.

6.2.3 Performance on uniformly distributed dataset. The lookup throughput and memory cost on skewed datasets with Zipfian and normal distributions demonstrate the exceptional advantages of STEM^2. One key reason is that the memory usage of our designed separator, XBSS, decreases as the size ratio between the two partitioned subgroups increases, as analyzed in Section 4.3. To further



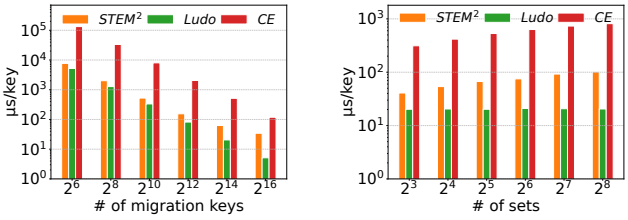
(a) Lookup thro. vs. # of keys (b) Memory cost vs. # of keys (c) Lookup thro. vs. # of sets (d) Memory cost vs. # of sets

Figure 8: Lookup Throughput and Memory Cost Using Datasets under Uniform Distribution.



(a) Amortized insert. latency vs. # of keys (b) Amortized insert. latency vs. # of sets

Figure 9: Amortized Insertion Latency Per Key.



(a) Migration latency vs. # of keys (b) Migration latency vs. # of sets

Figure 10: Amortized Migration Latency Per Key.

evaluate the applicability of STEM^2 in more general scenarios, we construct datasets with varying numbers of sets and total keys, where the set sizes for each dataset follow a uniform distribution, and examine STEM^2's performance on these datasets.

Lookup throughput and memory cost vs. number of keys. For the experimental setting, the number of sets is set as 2^5 and the number of keys varies from 2^{12} to 2^{26}. Figure 8(a) and Figure 8(b) show the lookup throughput and memory cost respectively. The results show that **STEM^2 achieves the highest lookup throughput** compared with other methods. **STEM^2 and Ludo incur the lowest memory cost, with STEM^2 requiring only 12.18–12.46 bits per key and Ludo requires 12.2 bits per key.** Notably, although TMSQ$_{FC}$ consumes 35.75 bits per key when the dataset size is 2^{12}, its memory efficiency improves significantly for larger datasets (e.g. 2^{18} to 2^{26}), requiring only an average of 11.26 bits per key. The reason TMSQ$_{FC}$ performs poorly in terms of memory cost on small datasets is the width of each bloom filter in the filter cascade is set with a fixed lower bound to prevent the number of layers from growing too rapidly. Consequently, even if the current Bloom filter produces only one false positive key, another Bloom filter layer must be added. In addition, the total number of keys is too small to amortize the relatively large memory cost.

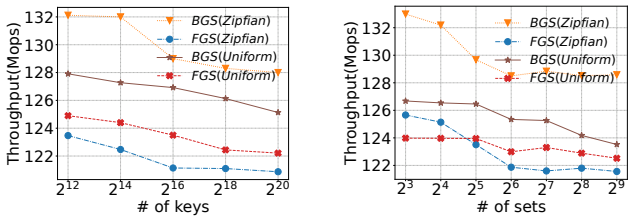
Lookup throughput and Memory cost vs. number of sets. We set the number of keys as 2^{24} and vary the number of sets from 2^3 to 2^9. As shown in Figure 8(c) and Figure 8(d), STEM^2 achieves over 120 Mops lookup throughput, **outperforming all other methods**. As the number of sets grows, the memory cost of STEM^2 increases from 7.31 to 21.92 bits per key. In contrast, Ludo requires an average of 12.2 bits to store a key, regardless of how the number of sets changes. TMSQ$_{FC}$ as the second-best benchmark, requires an average of 6.06–19.47 bits to store a key, which is very close to STEM^2.

Summary. STEM^2 achieves the highest lookup throughput across all datasets. It also incurs the lowest memory cost on skewed datasets under both Zipfian and normal distributions. Even on the uniformly distributed datasets—its worst case in terms of memory usage—STEM^2 exhibits only slightly higher memory overhead than Ludo and remains comparable to TMSQ$_{FC}$, while still outperforming all other baselines.

6.3 Control Plane Update Evaluation of STEM^2

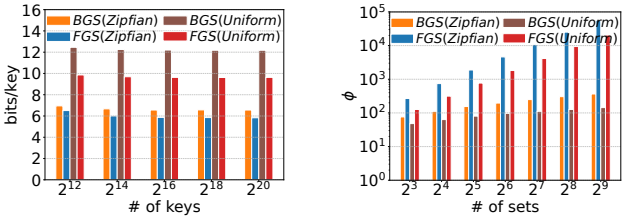
The control plane of STEM^2 is responsible for constructing the data structure and dynamically updating it if there are membership changes. Among these baselines, only Ludo, TMSQ$_{FC}$ and TMSQ$_{DASS}$ can provide 100% query accuracy. Since TMSQ$_{FC}$ and TMSQ$_{DASS}$ have the same structural framework as STEM^2, and STEM^2 outperforms them in prior evaluations, we select Ludo as the baseline for control plane update evaluation. In addition, to compare STEM^2 with the approximate data structures, we also include CE, which achieves the highest lookup throughput among evaluated approximate data structures. All evaluations in this subsection are conducted on datasets with Zipfian and uniform distribution.

Insertion latency vs. # of insertion keys. We evaluate the amortized insertion latency as the number of insertion keys increases. In the experimental setup, the original structure stores 2^{20} keys across 2^5 sets. We then insert more keys, ranging from 2^6 to 2^{16} in total, each insertion key is randomly assigned to one of the sets. As shown in Figure 9(a), Ludo is the fastest one in insertion among the three methods. STEM^2 incurs 1.31×–2.51× the amortized insertion latency compared to Ludo. CE is the slowest in insertion, causing 4.55×–18.81× the latency compared to STEM^2, as it does not support dynamic updates and requires a full rebuild upon each new insertion. For all methods, when the number of new keys increases, the amortized latency decreases. When 2^{14} keys are inserted, the amortized per-key insertion latency for STEM^2, Ludo and CE is 52.2 μs, 22 μs and 501 μs, while the latencies are reduced to 27.6 μs, 11 μs and 125.52 μs respectively for inserting 2^{16} keys. Additionally, both



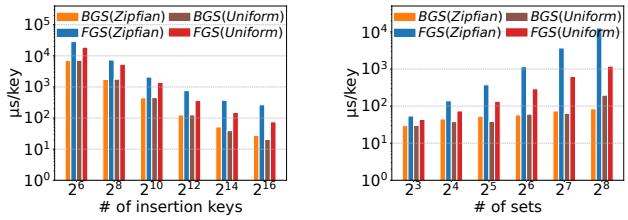
(a) Throughput vs. # of keys (b) Throughput vs. # of sets

Figure 11: Thro. Comparison Using Two Splitting Strategies.



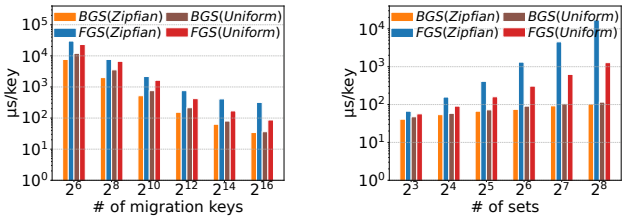
(a) DP mem. cost vs. # of keys (b) CP mem. cost vs. # of sets

Figure 12: Mem. Cost Using Two Splitting Strategies.



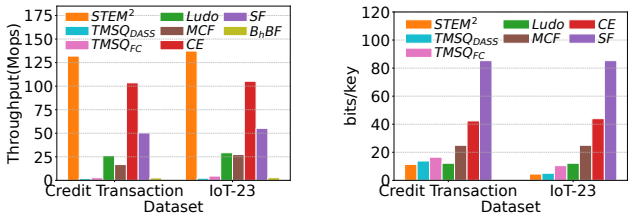
(a) Insertion latency vs. # of keys (b) Insertion latency vs. # of sets

Figure 13: Insertion Latency of Two Splitting Strategies.



(a) Migration latency vs. # of keys (b) Migration latency vs. # of sets

Figure 14: Migration Latency of Two Splitting Strategies.



(a) Throughput (b) Memory

Figure 15: Lookup Thro. and Memory Cost on Case Studies.

Ludo and STEM^2 rely on Othello structures, which may require rebuilding during insertions. Due to STEM^2 's adoption of hash function reuse across multiple XBSS, the probability of triggering an Othello rebuild is higher than that in Ludo. This design trade-off explains why Ludo achieves better update efficiency overall.

Insertion latency vs. number of sets. In this experimental setup, we fix the number of insertion keys to be 2^{14}. The original data structure stores a total of 2^{20} keys, with the number of sets varying from 2^3 to 2^8. Figure 9(b) shows the amortized insertion latency per key. When the number of sets grows, the amortized insertion time of STEM^2 grows from 30 μs to 85.46 μs, because more XBSSs need to be traversed during the insertion. In contrast, the amortized insertion time of Ludo remains stable at 23 μs, while that of CE increases from 314 μs to 746.21 μs. In summary, the batch insertion of STEM^2 is highly efficient, with an average per-key insertion latency remaining in the microsecond range.

We further evaluate key migration latency, which involves both insertion and deletion operations.

Key migration latency vs. number of keys. We evaluate the amortized migration latency that varies with the different numbers of migration keys. In the experimental setup, the original structure stores 2^{20} keys across 2^5 sets. We then randomly select a number of keys ranging from 2^6 to 2^{16}, with each selected key randomly migrated to another set. As shown in Figure 10(a), Ludo is the fastest one in migration, followed by STEM^2, and CE incurs the highest migration latency. Specifically, STEM^2 causes 1.46×–6.7× the migration latency of Ludo, while CE causes 3.42×–17.34× the migration latency of STEM^2. When 2^{16} keys are migrated, the amortized per-key migration latency for STEM^2, Ludo and CE is 34.5 μs, 5.15 μs and 118 μs respectively.

Key migration latency vs. number of sets. In this experimental setup, we fix the number of migration keys to be 2^{14}. The original data structure stores a total of 2^{20} keys, with the number of sets varying from 2^3 to 2^8. As shown in Figure 10(b), as the number of sets increases, the amortized key migration latency of STEM^2 increases from 41.13 μs to 102.1 μs, because more XBSS instances need to be traversed during the migration process. In contrast, the amortized migration latency of Ludo remains nearly constant at around 21 μs, while that of CE increases significantly from 314 μs to 811 μs. Comparing Figure 9(b) and Figure 10(b), we can find that the key migration process in STEM^2 involves both deleting the key from its original set and inserting it into the new set, which results in a longer delay compared to the pure insertion operation.

Discussion. Although STEM^2 has slightly higher update latency than Ludo, particularly when the number of sets is large, Ludo provides much lower query throughput. Thus, for query-intensive or latency-sensitive applications with infrequent updates, STEM^2 achieves a better trade-off.

6.4 Performance of Splitting Strategies

We evaluate two splitting strategies, FGS and BGS, for binary-tree construction on datasets with Zipfian and Uniform distributions.

Lookup throughput. We evaluate lookup throughput as the number of keys and sets varies. Figure 11(a) shows the results when the number of keys increases from 2^{12} to 2^{20} with the number of sets

fixed at 2^5, and Figure 11(b) shows the results when the number of sets increases from 2^3 to 2^9 with the number of keys fixed at 2^{20}. The results show BGS achieves 5.0%–7.8% higher lookup throughput than FGS on Zipfian-distributed datasets, and 1.1%–3.0% higher throughput on uniformly distributed datasets. Under FGS, a query with 2^9 sets traverses up to 512 XBSS instances, whereas BGS traverses only 9. Since less-hashing technique keeps the per-XBSS lookup overhead low, the throughput gap remains modest.

Data plane (DP) memory cost. We use the same experimental settings to evaluate data-plane memory cost under the two splitting strategies. Figure 12(a) shows the results when the number of sets is fixed at 2^5 and the number of keys increases from 2^{12} to 2^{20}. On Zipfian-distributed datasets, the memory cost decreases from 6.95 to 6.56 bits/key under BGS and from 6.51 to 5.83 bits/key under FGS, giving FGS a 6.3%–11.1% advantage. On uniformly distributed datasets, the memory cost decreases from 12.46 to 12.18 bits/key under BGS and from 9.86 to 9.63 bits/key under FGS, corresponding to a larger gap. In summary, FGS consistently achieves lower DP memory cost than BGS, and its advantage is modest on Zipfian datasets but substantially larger on uniformly distributed datasets. This is because the more aggressive greedy partitioning in FGS is especially beneficial when the set sizes are less skewed.

Control plane (CP) memory cost. As described in Section 4.1, STEM^2 introduces an index table for each counting Bloom filter to avoid re-testing all keys during update operations. This index table is maintained as part of the control plane. We adopt the same experimental settings as that in evaluating the lookup throughput and data plane memory cost. To quantify the control-plane memory overhead, we introduce a new metric, denoted by ϕ, defined as the ratio between the number of keys stored in the index table and the total number of keys. This metric captures the relative memory burden imposed by the index table, which incurs substantially higher memory cost than the core data structure of STEM^2.

As shown in Figure 12(b), when the number of keys is fixed at 2^{20} and the number of sets increases from 2^3 to 2^9: the index-table memory cost of FGS is 3.48×–157.18× that of BGS on Zipfian-distributed datasets and 2.65×–144.8× on uniformly distributed datasets. This gap arises because FGS increases the ratio $r = |D_L|/|D_S|$ for each XBSS in STEM^2, which results in a larger CBF index table. Besides, under the same strategy and dataset settings, uniformly distributed datasets incur lower CP memory overhead than Zipfian datasets.

New key insertion latency. Figure 13 compares the insertion latency of the two splitting strategies. Because FGS introduces more levels in the binary tree, its average per-key insertion latency is substantially higher than that of the BGS strategy used in STEM^2. Specifically, when inserting 2^6–2^{16} new keys into STEM^2 with 2^5 sets and an initial size of 2^{20} keys, FGS incurs 4.06×–9.60× higher insertion latency than BGS on Zipfian-distributed datasets, and 2.66×–3.84× higher latency on uniformly distributed datasets.

When STEM^2 initially stores 2^{20} keys and the number of sets increases from 2^3 to 2^8, inserting 2^{14} new keys into the Zipfian dataset causes FGS to incur 1.82×–149.59× the insertion latency of BGS, indicating a substantial gap. On uniformly distributed datasets, the corresponding gap is smaller but still significant, with FGS incurring 1.43×–9.79× higher insertion latency than BGS.

Key migration latency. We use the same experimental setup as in the key insertion latency evaluation. As shown in Figure 14, FGS consistently incurs higher migration latency than BGS. When migrating 2^6–2^{16} random keys in STEM^2 with 2^5 sets and an initial size of 2^{20} keys, the amortized migration latency of FGS is 3.84×–9.24× that of BGS on Zipfian-distributed datasets, and 1.87×–2.41× on uniformly distributed datasets. When the original structure stores 2^{20} keys and the number of sets increases from 2^3 to 2^8, the gap becomes even larger. On Zipfian-distributed datasets, the migration latency of FGS is 1.61×–165.41× that of BGS. On uniformly distributed datasets, FGS still incurs 1.2×–10.98× higher migration latency. This difference arises because FGS produces a deeper and more unbalanced tree, causing each migration to update many more XBSS instances, and the penalty becomes especially pronounced as the number of sets increases.

Discussion. Overall, FGS and BGS expose different trade-offs. FGS achieves lower DP memory cost, with the advantage being especially noticeable on uniformly distributed datasets, but it incurs substantially higher CP memory overhead and update latency due to its deeper and more unbalanced tree structure. In contrast, BGS provides higher lookup throughput and much better update efficiency, while incurring only a modest increase in data-plane memory cost.

6.5 Evaluation on Real Dataset

We evaluate the performance of STEM^2 in two case studies using the aforementioned network traffic and credit card transaction datasets. Figure 15 shows the lookup throughput and memory cost. The results show that **STEM^2 achieves the highest lookup throughput** among all methods, exceeding 130 Mops. For memory cost, STEM^2 requires an average of 11.41 and 4.49 bits per key, respectively, achieving the lowest memory cost among all methods.

7 CONCLUSION

This paper presents a holistic design for exact multi-set membership queries, centered on the proposed STEM^2 data structure. Through a decoupled architecture, STEM^2 separates maintenance and query processing into two planes: a maintenance-oriented control plane for structure construction and incremental updates, and a query-optimized data plane for high-throughput lookups. This design is enabled by our novel Exact Binary Set Separator (XBSS) and the less-hashing technique that requires only two hash computations, jointly ensuring 100% query accuracy without sacrificing performance. Extensive experiments show that STEM^2 outperforms state-of-the-art hashing- and filter-based solutions in both throughput and memory efficiency, providing a robust and scalable foundation for high-performance networking and database applications.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant IUCRC-1916756 and by industry funding from the Center for Hardware Embedded Systems Security and Trust (CHEST). The work of Yannian Niu and Minmei Wang was partially supported by the National Science Foundation under Grant CNS-2426030. The work of Song Han was partially supported by the National Science Foundation under Grant CNS-2008463.

REFERENCES

[1] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An improved construction for counting bloom filters. In *European Symposium on algorithms*. 684–695.

[2] Alex D Breslow and Nuwan S Jayasena. 2018. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment* (2018), 1041–1055.

[3] Andrei Broder and Michael Mitzenmacher. 2004. Network applications of bloom filters: A survey. *Internet mathematics* 1, 4 (2004), 485–509.

[4] Denis Charles and Kumar Chellapilla. 2008. Bloomier filters: A second look. In *European Symposium on Algorithms*. Springer, 259–270.

[5] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*. 30–39.

[6] Hanhua Chen, Liangyi Liao, Hai Jin, and Jie Wu. 2017. The dynamic cuckoo filter. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. IEEE, 1–10.

[7] Lin Chen and Jihong Yu. 2021. Multiset membership lookup in large datasets. *IEEE Transactions on Knowledge and Data Engineering* 34, 10 (2021), 4947–4958.

[8] Haipeng Dai, Yuankun Zhong, Alex X Liu, Wei Wang, and Meng Li. 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.

[9] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. 2014. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 75–88.

[10] Bin Fan, Dong Zhou, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2013. When cycles are cheap, some tables can be huge. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*.

[11] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. 2000. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM transactions on networking* (2000), 281–293.

[12] Daniel Flachs, Magnus Müller, and Guido Moerkotte. 2022. The 3D hash join: building on non-unique join attributes. CIDR.

[13] Pengtao Fu, Lailong Luo, Deke Guo, Shangsen Li, and Yun Zhou. 2023. Shifting filter code. https://github.com/fptjy/Shifiing-filter-framework.

[14] Pengtao Fu, Lailong Luo, Deke Guo, Shangsen Li, and Yun Zhou. 2023. A shifting filter framework for dynamic set queries. *IEEE/ACM Transactions on Networking* (2023), 2329–2344.

[15] Sebastian Garcia, Agustin Parmisano, and Maria Jose Erquiaga. 2020. IoT-23: A labeled dataset with malicious and benign IoT network traffic. Data set. https://doi.org/10.5281/zenodo.4743746

[16] Google. 2014. FarmHash: Hash functions for strings. https://github.com/google/farmhash. Accessed: October 21, 2025.

[17] Adam Kirsch and Michael Mitzenmacher. 2006. Less hashing, same performance: Building a better bloom filter. In *European Symposium on Algorithms*. 456–467.

[18] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. 2017. CRLite: A scalable system for pushing all TLS revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy (SP)*. 539–556.

[19] Ryan Lee and Priyam Choksi. 2024. *Credit Card Transactions Dataset.* https://www.kaggle.com/datasets/priyamchoksi/credit-card-transactions-dataset Data set.

[20] Rundong Li, Pinghui Wang, Jiongli Zhu, Junzhou Zhao, Jia Di, Xiaofei Yang, and Kai Ye. 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.

[21] Hyeontaek Lim, Bin Fan, David G Andersen, and Michael Kaminsky. 2011. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 1–13.

[22] Peng Liu, Hao Wang, Siang Gao, Tong Yang, Lei Zou, Lorna Uden, and Xiaoming Li. 2018. ID bloom filter: Achieving faster multi-set membership query in network applications. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–6.

[23] Qiyu Liu, Libin Zheng, Yanyan Shen, and Lei Chen. 2020. Stable learned bloom filters for data streams. *Proceedings of the VLDB Endowment* (2020), 2355–2367.

[24] Caroline Lo, Dan Frankowski, and Jure Leskovec. 2016. Understanding behaviors that lead to purchasing: A case study of pinterest. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 531–540.

[25] Lailong Luo, Deke Guo, Yawei Zhao, Ori Rottenstreich, Richard TB Ma, and Xueshan Luo. 2021. MCFsyn: A multi-party set reconciliation protocol with the marked cuckoo filter. *IEEE Transactions on Parallel and Distributed Systems* 32, 11 (2021), 2705–2718.

[26] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. 2018. Adaptive Cuckoo Filters. In *2018 Proceedings of the Meeting on Algorithm Engineering and Experiments (ALENEX)*. 36–47.

[27] Ofri Nevo, Ni Trieu, and Avishay Yanai. 2021. Simple, fast malicious multiparty private set intersection. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1151–1165.

[28] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[29] Shuyu Pei, Kun Xie, Xin Wang, Gaogang Xie, Kenli Li, Wei Li, Yanbiao Li, and Jigang Wen. 2022. BhBF: A Bloom Filter Using B h Sequences for Multi-set Membership Query. *ACM Transactions on Knowledge Discovery from Data (TKDD)* (2022), 1–26.

[30] Subhabrata Sen and Jia Wang. 2002. Analyzing peer-to-peer traffic across large networks. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*. 137–150.

[31] Shouqian Shi and Chen Qian. 2020. Ludo hashing code. https://github.com/QianLabUCSC/Ludo.

[32] Shouqian Shi and Chen Qian. 2020. Ludo hashing: Compact, fast, and dynamic key-value lookups for practical network systems. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* (2020), 1–32.

[33] Shouqian Shi, Chen Qian, and Minmei Wang. 2019. Re-designing compact-structure based forwarding for programmable networks. In *2019 IEEE 27th International Conference on Network Protocols (ICNP)*. IEEE, 1–11.

[34] Xiaofeng Shi, Shouqian Shi, Minmei Wang, Jonne Kaunisto, and Chen Qian. 2021. On-device IoT certificate revocation checking with small memory and low latency. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1118–1134.

[35] Zhouyi Sun, Siang Gao, Bingqing Liu, Yufei Wang, Tong Yang, and Bin Cui. 2019. Magic cube bloom filter: Answering membership queries for multiple sets. In *2019 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 1–8.

[36] Yang Tong, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. 2019. Coloring embedder: A memory efficient data structure for answering multi-set query. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1142–1153.

[37] Yang Tong, Dongsheng Yang, Jie Jiang, Siang Gao, Bin Cui, Lei Shi, and Xiaoming Li. 2019. Coloring embedder code. https://github.com/4colorclassifier/4colorclassifier.

[38] Minmei Wang and Mingxun Zhou. 2019. Vacuum filters: more space-efficient and faster replacement for bloom and cuckoo filters. *Proceedings of the VLDB Endowment* (2019), 197–210.

[39] Zhuohan Xie, Wencheng Ding, Hongya Wang, Yingyuan Xiao, and Zhenyu Liu. 2017. D-ary cuckoo filter: A space efficient data structure for set membership lookup. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, 190–197.

[40] Dongsheng Yang, Deyu Tian, Junzhi Gong, Siang Gao, Tong Yang, and Xiaoming Li. 2017. Difference bloom filter: A probabilistic structure for multi-set membership query. In *2017 IEEE International Conference on Communications (ICC)*. 1–6.

[41] Tong Yang, Alex X. Liu, Muhammad Shahzad, Yuankun Zhong, Qiaobin Fu, Zi Li, Gaogang Xie, and Xiaoming Li. 2016. A Shifting Bloom Filter Framework for Set Queries. *Proc. VLDB Endow.* (2016), 408–419.

[42] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. 2014. Guarantee IP lookup performance with FIB explosion. In *Proceedings of the 2014 ACM Conference on SIGCOMM*. 39–50.

[43] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: Bloom filter forwarding architecture for large organizations. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. 313–324.

[44] Ye Yu, Djamal Belazzougui, Chen Qian, and Qin Zhang. 2018. Memory-efficient and Ultra-fast Network Lookup and Forwarding using Othello Hashing. *Proc. of IEEE/ACM Transactions on Networking* (2018), 1151–1164.

[45] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. 2023. Linear private set union from {Multi-Query} reverse private membership test. In *32nd USENIX Security Symposium (USENIX Security 23)*. 337–354.

[46] Kai Zhang, Kaibo Wang, Yuan Yuan, Lei Guo, Rubao Lee, and Xiaodong Zhang. 2015. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment* (2015), 1226–1237.

[47] Dong Zhou, Bin Fan, Hyeontaek Lim, David G Andersen, Michael Kaminsky, Michael Mitzenmacher, Ren Wang, and Ajaypal Singh. 2015. Scaling up clustered network appliances with ScaleBricks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 241–254.