Coloring Embedder: Towards Multi-Set Membership Queries in Web Cache Sharing

Zhaodong Kang[†], Jin Xu[¶], Wenqi Wang[*], Jie Jiang[*], Shiqi Jiang[††], Tong Yang[*‡], Bin Cui[*§], Tilman Wolf[‖]
[†]School of Electronic and Computer Engineering, Peking University, China
[¶]Yuanpei College, Peking University, China [*]Department of Computer Science, Peking University, China
[§]National Engineering Laboratory for Big Data Analysis Technology and Application, Peking University, China
[‡]PCL Research Center of Networks and Communications, Pengcheng Laboratory
[††]School of Mathematical Sciences, Peking University, China [‖]University of Massachusetts Amherst, USA

***Abstract*—Multi-set membership queries are fundamental operations in data science. In this paper, we propose a new data structure for multi-set membership queries, named coloring embedder, which is fast, accurate, and memory efficient. The idea of coloring embedder is to first map elements to a high-dimensional space, which nearly eliminates hashing collisions, and then use a dimensional reduction representation, similar to coloring a graph, to save memory. Theoretical proofs and experimental results show that the coloring embedder is effective in solving the problem of multi-set membership queries. We also find that web cache sharing is one of the typical application scenarios of the multi-set membership queries and current methods based on Bloom filters always send redundant queries. We apply coloring embedder to web cache sharing by arranging our data structure on the on-chip and off-chip memory and designing query, insertion and deletion operations for this scenario. The experimental results show that compared with the present method, our method can reduce the queries sent by proxies while reaching equal hit rate with the same size of on-chip memory. The source code of coloring embedder has been released on Github.**

I. Introduction

Multi-set membership queries aim to find which set an element belongs to. A multi-set membership query is a fundamental operation in computer science. This type of query appears in many applications, including indexing in data centers [40], [52], distributed file system [5], database indexing [5], data duplication [31], network packets processing [12], [46], [48], and network traffic measurement [13], [44]. Before discussing existing approaches to solving multi-set membership queries and the novelty of our approach, we briefly provide a formal definition and two example use cases.

A. Formal Definition

Given s sets $\mathcal{S}_1$, $\mathcal{S}_2$ $\ldots\mathcal{S}_s$ with no intersection and an element e from one of those sets, a multi-set membership query returns the set which e belongs to. The formal definition is as follow.

Multiset Membership Query: U is the universe of elements, *i.e.*, $U = \{e_1, e_2, ...e_i..., e_m\}$, where e_i can be a string, an integer, an IP address, etc. U is partitioned into s disjoint sets $S_1, S_2, ..., S_s$, such that $\forall i, j, S_i \cap S_j = \emptyset$, and $S_1 \cup S_2 \cup ... \cup S_s = U$. The membership of e can be represented by a function $f : U \mapsto \{1, 2, ..., s\}$, such that $f(e) = i$ if $e \in S_i$, where i is also defined as the set ID of e. For any element $e \in U$, the multi-set membership query is to retrieve its set ID, denoted as $\hat{f}(e)$, which is the result produced by the query and may differ from $f(e)$.

Our goal is to design an algorithm for multi-set membership queries, which encodes f into a data structure D, and answer queries based on D. If the answer $\hat{f}(e)$ for querying e is not equal to $f(e)$, we say this query incurs an *error*. In practice, a few errors are typically acceptable, especially in big data scenarios.

Corresponding author: Tong Yang (Email: yangtongemail@gmail.com).
The first two authors contribute equally.

B. Example Use Cases and Performance Metrics

To illustrate the use of multi-set membership queries, we provide two typical use cases:

Use Case 1: Distributed caching. A classic distributed caching approach is the Summary Cache [22]. There are multiple proxy caches, and each proxy keeps a compact summary of the cache content of every other proxy cache. When a cache miss occurs, the cache first checks all the summaries to see if the request might be a hit in other caches, and then sends the query message to those proxies whose summaries show positive results. This is a typical multi-set membership query problem. Due to the importance of distributed caching, recent work [49], [51] still aims to optimize its performance.

Use Case 2: MAC table query. In data centers, switches need to determine the outgoing port for each incoming packet. Switches query the MAC table to find the output port information, which can be seen as a multi-set membership query. Each MAC table entry includes a key (MAC address) and a value (port). A typical MAC table [2] contains around ten thousand entries and tens of ports. However, switches often have limited memory, making it challenging to support queries at high line-rates [48]. Existing solutions [36], [48] sacrifice query accuracy, which means that a query may lead to a wrong answer (error). In the case of MAC tables, such errors are acceptable, but may incur high time penalties.

The key metrics of multi-set membership queries are query speed, error rate, and memory usage. High query speed is critical to achieve a high throughput of query requests [39]. Low error rate is highly desirable because the penalties for errors

may be high. Small memory usage is also important, because processor caches are typically small, and data structures should be small enough to fit into cache to achieve fast access speeds [27]. Prior work has often focused on improving one or two of these three metrics. The objective of our paper is to optimize all three metrics at the same time.

C. Prior Art and Limitations

In general, there are two types of solutions for multi-set membership query: hash-table-based solutions and Bloom-filter-based solutions.

Using a hash table is a straightforward solution for the multi-set membership query problem. We use elements as keys and the set identifiers as values. We then can build a hash table based on these key-value pairs. Hash-table-based solutions are accurate but not memory-efficient. Traditional hash-table-based solutions [33] achieve O(1) query speed at the cost of large memory usage. However, if the memory is limited, the query time will become unbounded due to hash collisions.

Solutions using perfect hashing [10], [15] sacrifice insertion speed for query speed and have bounded query time. However, they do not support fast dynamic updates. Cuckoo hashing [34] and cuckoo filter [21] can achieve high query speed, but it uses memory to store the fingerprint of keys. The memory usage may be large when a high accuracy is desired.

A Bloom filter [8] is a compact data structure for membership query problem. Bloom filters can achieve fast and constant query speed using very small memory at the cost of sacrificing query accuracy. A large body of work [13], [32], [43], [45] uses Bloom filters for the multi-set membership query problem. However, that work suffers from a relatively high error rate because of hashing collisions. When an element fully overlaps with another elements in a Bloom filter, a false positive happens.

None of the above solutions excels in all three key metrics that are relevant for multi-set membership queries, i.e., fast query speed, low error rate, and low memory use.

D. Proposed Approach

In this paper, we propose a novel data structure, named the *coloring embedder*, which can achieve fast query speed, almost no error, and low memory usage at the same time.

Similar to hash-table-based solutions and Bloom-filter-based solutions, our coloring embedder is also based on hashing.

Before introducing our solution, let us consider the following scenario: Given m elements, assume they are randomly mapped to $n = cm$ buckets. (In this paper, a bucket means a unit of memory that can store exactly one element.) An element cannot be represented by its bucket if two or more different elements are mapped to this bucket, and we call such case a collision. Clearly, many collisions will occur when $c = 1$. To reduce the number of collisions to a practical level, c (and the necessary memory) has to be very large.

The approach of the coloring embedder is to allow a very small number of collisions with limited memory overhead. There are two challenges to design such a data structure: one is how to map the elements to eliminate collisions, and the other is how to use small amounts of memory to store the mapping results. To handle the above challenges, we propose two key techniques: The first one is hyper-mapping and the second one is coloring embedding.

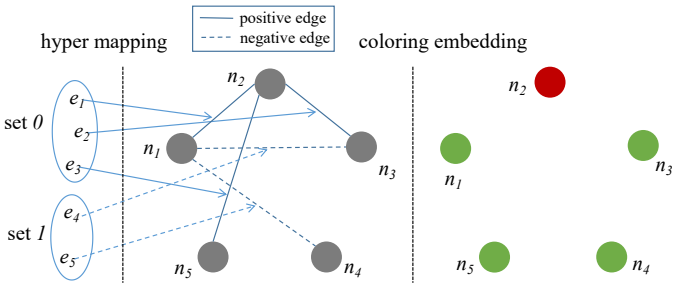


Fig. 1: Example of hyper-mapping and coloring embedding.

We first map all elements to a high-dimensional space to almost eliminate hashing collisions. Then we perform dimensionality reduction to map the high-dimension space to a low-dimension space.

We use graph terminology to explain our algorithm. Suppose there are m elements, we first map them to an empty graph with cm nodes and $(cm/2)^2$ edge slots, where c is recommended to be 2.2 according to our experimental results. (See Section VI-B) Each element is mapped to an edge slot to build an edge and the set ID is recorded on the edge. Then, we embed the graph with $(cm/2)^2$ edge slots into a node vector with cm nodes, while keeping the recorded set identifiers of all elements accurate.

We propose to use the colors of the nodes to represent the type of the edges, namely coloring embedding.

To illustrate the principle of coloring embedding, we consider a simple case with only two sets, set 0 and set 1. For convenience, we name edges mapped by elements in set 0 as positive edges, and edges mapped by elements in set 1 as negative edges. The graph is colored according to two `coloring rules`:

1) If there is a positive edge between any two nodes, those two nodes should have different colors;
2) If there is a negative edge between any two nodes, they should have the same color.

If all nodes can be colored according to the two rules, the coloring embedding succeeds. Then, we can answer a multi-set membership query with only the vector of node colors. When an element in the sets is queried, we check the two end-nodes of its mapped edge. If the two nodes have different colors, the element is in set 0. Otherwise, it is in set 1.

For more than two sets, we do not directly encode the set identifiers in pairs. Instead, we encode the set identifiers by bits. If there are totally s sets, we can encode 0 to $s-1$ with no more than $\lceil log(s) \rceil$ bits using binary coding. Each bit can be represented by an edge, then the length of the set ID is $\lceil log(s) \rceil$ bits. (See Section III-C) As mentioned above,

one graph can encode two sets, so one graph can encode the content of one bit. Therefore, we can create $\lceil log(s) \rceil$ graphs and each graph encodes one bit of the binary representation of a set ID. To achieve faster query speed and better load balancing, we further propose an approach called shifting coloring embedder, which uses only one graph. More details are discussed in Section IV.

II. Background and Related Work

In this section, we discuss related work for multi-set queries and introduce properties of random graphs, which are relevant to our proposed algorithm.

A. Exact-match Data Structures

Exact-match data structures that are based on hash tables [33] have no error. However, they need to deal with hash collisions and therefore store element keys in order to resolve these conflicts. To reduce the hash collision rate to support fast queries and updates, a large memory is needed. Perfect hashing [7], [19] requires little memory redundancy, but cannot easily support insertions.

Cuckoo hashing [34] and cuckoo filter [21] can achieve high query speed. And [38] is an example of its application scenario. Cuckoo hashing maps each element to two positions. If both positions are occupied by other elements, the algorithm expels one of those elements to make room for the new element and inserts the expelled element to its other position. but it needs to store the fingerprints of keys. When load factor is high, a cuckoo hashing update could also fail.

B. Probabilistic Data Structures

Probabilistic data structures for multi-set query are commonly based on Bloom filters [8]. A Bloom filter is a compact data structure to represent a set and supports approximate membership query, i.e., answering whether an element belongs to the set, but the answer may be wrong. A Bloom filter consists of a bit array and k hash functions that map an element to k bits in the array. To insert an element, k hash functions are computed and all the mapped k bits are set to 1. To query an element, the Bloom filter checks the k mapped bits and returns true if and only if all of them are 1.

A straightforward method for multi-set queries is to use multiple Bloom filters, each recording one set [48]. But this method has low memory efficiency and slow query speed because it needs to access multiple Bloom filters. Recent work has aimed to reduce the number of Bloom filters by letting each Bloom filter represent a part of the encoded set IDs, such as Bloom Tree [47], Coded Bloom filter [13], Sparsely coded filter [32]. Since the optimal length of a Bloom filter is related to the number of elements, the memory usage of these methods may be influenced by the distribution of set sizes, even if the total number of elements is given.

There are also Bloom filter variances that record elements of different sets in a single filter, such as the Combinatorial Bloom filter [25], iSet [36], the Shifting Bloom filter [45], and more [16]–[18], [44]. They share the advantage that they are not influenced by the distribution of set sizes.

TABLE I: Notation used in this paper.

Symbol	Description
m	# edges or # elements
n	# nodes or # buckets
e	an element
$n/m\ ratio$	n divided by m
s	# sets
$h(.)$	hash functions
S^+	the 1^{st} set of elements
S^-	the 2^{nd} set of elements
m^+	# elements in S^+ or # positive edges
m^-	# elements in S^- or # negative edges

Bloom filters are suitable for the scenario where the allowed error rate is relatively high. If the allowed error rate is very low (e.g., 10^{-4}), Bloom filters need too much memory (e.g., 19.13 bits per element using 13 hash functions) to reduce the collision rate to meet this requirement. By contrast, our algorithm has the property that if the memory is above a rather small threshold (2.2 bits per element using 2 hash functions), there are almost no errors at all (less than 10^{-4} for 10^5 elements). Thus, our algorithm is much more memory efficient for low-error-rate scenarios.

C. Random Graph and Sharp Threshold

A random graph is generated by randomly connects m pairs of nodes in an empty graph containing n nodes. Random graphs have many elegant mathematical properties. A typical one is the existence of *sharp threshold* [9], [50]. The sharp threshold is also called *phase transition phenomenon*, which means some properties may suddenly change when an independent variable is changed. For example, it has been proved that, cycles exist in a random graph with high probability when m/n is larger than 1, and there are no cycles with high probability when m/n is smaller than 1 [20]. Therefore, 1 is the sharp threshold of the existence of cycles. We have found that there also is a sharp threshold of memory for successful construction of the coloring embedder. The construction will succeed with high probability when the memory size is larger than the threshold. And there is also a sharp threshold of the load rate which influences the throughput of insertion. This property can be used for choosing a proper initial memory size for a coloring embedder.

III. The Coloring Embedder

In this section, we describe the design of the coloring embedder in detail. We first describe the coloring embedder for two-set queries and then present two variances for multi-set query. Table I summarizes the notation used in this paper.

A. Operation of Two-Set Query

The key idea of the coloring embedder is to *first map all elements to a high dimensional space to avoid hashing collisions and then perform dimensional reduction to embed the high dimension space into a low dimension space.* There are two steps to construct a coloring embedder, hyper mapping and coloring embedding, as illustrated in Figure 1.

TABLE II: Color for each state of the bucket.

Bits	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
Color	Red	Green	Blue	Yellow

In the hyper mapping process, we first build an empty graph with cm nodes, where m is the number of elements and c is a constant. Then, we map each element to an edge slot randomly using hash functions. Since there are about $(cm/2)^2$ edge slots, collisions rarely happen. We record the set IDs on the edges. For the two-set query, there are two sets: set 0 ($\mathcal{S}^+$) and set 1 ($\mathcal{S}^-$), and thus there are two kinds of edges in the graph. The edge with set ID 0 is named as *positive edge*, and the edge with set ID 1 is named as *negative edge*.

In the coloring embedding process, we embed the graph into a node vector by coloring the nodes in the graph. The coloring rules are: 1) for each positive edge, the colors of its two associated nodes should be different; 2) for each negative edge, the colors of its two associated nodes should be the same.

After all elements are mapped to the edges, we apply a RDG coloring algorithm to this graph. We introduce the algorithm later in Section III-B3. As our algorithm is randomized, we refer to the success rate of coloring embedding as the possibility of coloring the mentioned graph successfully using our algorithm. It is obvious that the more colors are used in our algorithm, the higher success rate it will reach and the more memory is used. In addition, using an integer power of 2 as the number of colors brings convenience to encoding and decoding because we can use adjacent bits to represent the color and save and read the color by shifting operation. If we use other digits such as 3, 5 or 6, we must use multiple to save and division operations to read the color. To balance the success rate of coloring embedding and memory usage, we finally use four colors to color the graph. Four colors have been proven enough according to the theoretical analysis in Section V.

If two nodes are connected by both negative and positive edges during constructing the graph, the collision error occurs. We ignore the positive edge in this case, which lead to errors in querying. When the constructed graph has collision errors, we generally allow the errors to exist because the error rate is always acceptably low. The detailed analysis can refer to Section V.

B. Implementation of Two-Set Query

Here, we describe the data structure and the operations in the coloring embedder, including construction, query, insertion, deletion, and migration.

1) Data Structure: The coloring embedder consists of two parts: a node array and an adjacency list. As shown in Figure 2, these two parts can be stored separately because they are used in different situations. Below, we describe each of them.

- Node Array: The node array is used to store the results of the coloring embedding. A node array consists of n buckets, and each bucket consists of two bits denoted by b_1 and b_2. Each bit can be set to 0 or 1, so a bucket has 4 states: (0,0), (0,1), (1,0), and (1,1), corresponding to four colors, red, green, blue, and yellow, respectively (see Table II).
  A bucket in the node array corresponds to a node in the graph. We define coloring a bucket as setting the values of the two bits in a bucket. For example, if we color a bucket with green, it means setting its first bit to 0 and its second bit to 1.
- Adjacency List: The adjacency list is used to store the edges of the graph during the hyper-mapping process. The list is composed of n linked lists, and the header of each linked list corresponds to a bucket in the node array. Let n_i denote the i^{th} bucket. If two nodes in the graph are connected by an edge, the two corresponding buckets in the node array are *logically adjacent*. The linked list of the i^{th} bucket stores the positions of all the buckets that are logically adjacent to bucket n_i. For each item in the linked list, we use a flag bit to indicate whether the edge is positive or negative.
  In Figure 2, positive edges are represented by solid lines, and negative edges are represented by dash lines. From the adjacency list in Figure 2 we can see that n_3 is logically adjacent to n_6 with a negative edge, and is logically adjacent to n_4 with a positive edge.

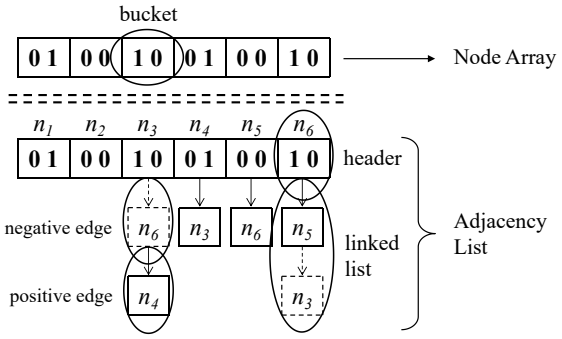


Fig. 2: Structure of the coloring embedder.

2) Operations: The operations in the coloring embedder are construction, query, insertion, deletion, and migration.

Construction: Initially, there is a node array with n buckets and a graph with n nodes and no edges. The i^{th} bucket with two bits corresponds to the i^{th} node with four colors, and we use n_i to denote them both. For each element e in $\mathcal{S}^+$ and $\mathcal{S}^-$, we compute two hash functions to map the element to two nodes $n_{h_1(e)}$, $n_{h_2(e)}$, and we create an edge between these two nodes. If the element is in set $\mathcal{S}^+$, the edge is a positive edge, otherwise it is a negative edge. After all elements are inserted, we color the graph to make all nodes obey the coloring rules mentioned in Section I-D. Any coloring algorithm can be used, and we present an algorithm named RDG in Section III-B3. If the graph is colored successfully, we assign the value of bucket n_i with the color of node n_i ($1 \leqslant i \leqslant n$). Otherwise, we change hash functions and repeat construction until it succeeds. Simply, we can change the seeds of the hash functions. When the memory size of the node array is larger than 2.2 bits per element so that the number of nodes exceeds the threshold, the construction will succeed in the first attempt with high probability (See Section VI-B).

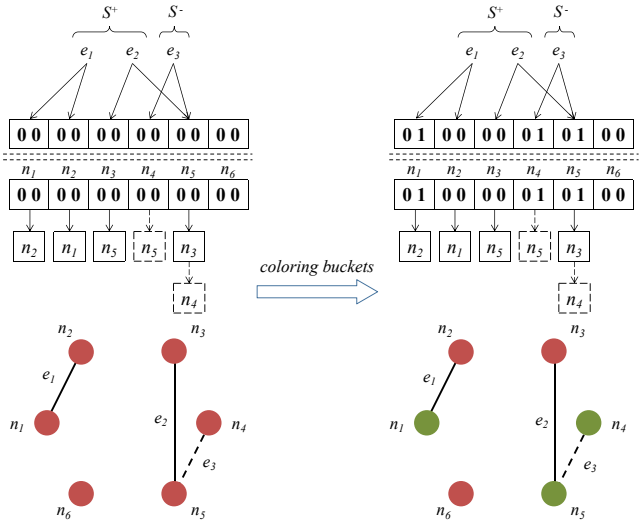


Fig. 3: An example of construction.

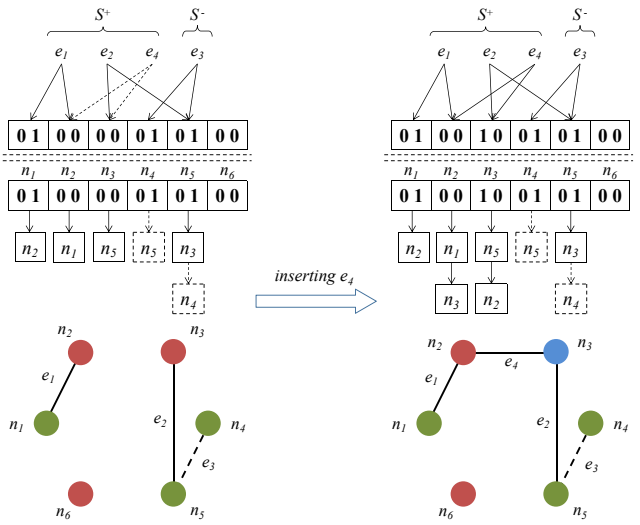


Fig. 4: An example of insertion.

Example (Figure 3): Set $\mathcal{S}^+$ has two elements, e_1 and e_2, and set $\mathcal{S}^-$ has one element, e_3. First, every element is mapped to the adjacency list and three logical edges are created. Two of the them are positive and one is negative. The three edges are showed in the corresponding graph below the coloring embedder. Positive edges are represented by solid lines and negative edges are represented by dash lines. Second, we color the nodes. As shown in the graph on the right, n_1 and n_2, n_3 and n_5 are colored with different colors; n_4 and n_5 are colored with the same color. Two colors are enough to color the graph. After that, we set the values of the buckets in both the node array and the adjacency list according to the color of the graph.

Query: The query process only involves the node array. When querying an element e, we compute the two hash functions for e and check the colors of the two mapped buckets $n_{h_1(x)}$ and $n_{h_2(x)}$. If the colors of the buckets are different, e belongs to $\mathcal{S}^+$. Otherwise, e belongs to $\mathcal{S}^-$.

Example (Figure 3): When querying element e_1, we compute hash functions and get two buckets n_1 and n_2 with values (0,1) and (0,0), respectively. Since these two buckets have different colors, the edge between them is a positive edge. Thus we report that e_1 belongs to $\mathcal{S}^+$.

Insertion: There are two steps to insert an element e. First, we compute the two hash functions and map e to two buckets $n_{h_1(x)}$ and $n_{h_2(x)}$. If e belongs to $\mathcal{S}^+$, we add a positive edge between the two buckets in the adjacency list; if e belongs to $\mathcal{S}^-$, we add a negative edge in the adjacency list. Second, we perform the *RDG updating* algorithm to make all affected buckets follow the coloring rules.

Example (Figure 4): A new element, e_4, from $\mathcal{S}^+$ will be inserted. First, we map e_4 to two buckets and add a positive edge between n_2 and n_3. Second, we find out that the colors of bucket n_2 and n_3 are both red, while their colors should be different according to the coloring rules. Therefore, we need to perform the RDG updating algorithm. As a result, the color of bucket n_3 changes from red to blue.

Deletion: To delete an element e, we compute the two hash functions to locate the buckets of e, and then remove the edge between $n_{h_1(x)}$ and $n_{h_2(x)}$ from the adjacency list. That means deleting $n_{h_1(x)}$ from the linked list of $n_{h_2(x)}$ and deleting $n_{h_2(x)}$ from the linked list of $n_{h_1(x)}$. The node array does not need to be modified at once.

Migration: Migration means an element e changes its membership from $\mathcal{S}^+$ to $\mathcal{S}^-$ or vice versa. If e migrates from $\mathcal{S}^+$ to $\mathcal{S}^-$, the edge between $n_{h_1(x)}$ and $n_{h_2(x)}$ changes from positive to negative; if e migrates from $\mathcal{S}^-$ to $\mathcal{S}^+$, the edge changes from negative to positive. Then, the RDG update algorithm is used to color other affected nodes.

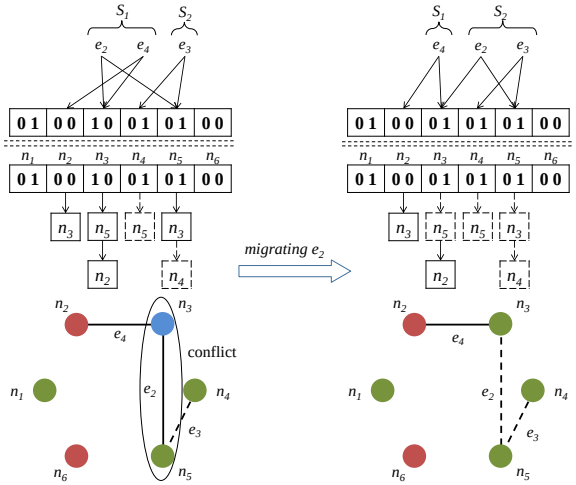


Fig. 5: An example of migration.

Example (Figure 5): Element e_2 changes its membership from $\mathcal{S}^+$ to $\mathcal{S}^-$ and the edge between n_3 and n_5 needs to be changed from positive to negative. As a result, we need to change the colors of n_3 and n_5 to be the same. The RDG update algorithm is performed, and the color of n_3 changes from blue to green. Other buckets are not affected in this case.

3) The RDG Coloring Algorithm: Here, we describe our coloring algorithm in details. The coloring problem is a well-known NP-complete problem [23] and no polynomial-time

exact algorithm exists. Our coloring algorithm, called *Recursively Delete or Give up coloring (RDG)*, is an extension of the k-core decomposition algorithm in [6]. It is a randomized algorithm that gives an approximation solution and is fast and accurate in practice.

Before going into the details of the algorithm, we introduce a well-known term in graph theory, k-core [14], [26], [37]: The k-core is the maximum subgraph in which the degree of every node is equal or larger than k. Our RDG algorithm is based on the observation that *the graph can be quickly and successfully colored with k colors if there is no k-core in the graph.*

For convenience, we use *CSG* to denote *Connected Sub-Graph*. Our RDG coloring algorithm is divided into the following steps:

1) For every pair of nodes directly connected by negative edges, we merge those two nodes to a single node. One new single node may be merged by more than 2 nodes. Collision errors occur when the new single node contains two nodes that have been connected by a positive edge. We just ignore that positive edge in this algorithm. After that, the graph only contains positive edges. A stack is built to record deleted nodes.
2) If all CSGs in the graph have been deleted, go to Step 5. Otherwise, for each CSG_i that is still not deleted, we compare its number of nodes N_{CSG_i} with the predefined threshold θ. If $N_{CSG_i} \leqslant \theta$, go to Step 3; If $N_{CSG_i} > \theta$, go to Step 4. Typically, we set θ to 16.
3) The incoming CSG is small, so we simply use a depth-first method to color it. If the coloring succeeds, we delete the CSG and return to Step 2. Otherwise, we report that the graph cannot be colored with four colors and the algorithm ends.
4) For the incoming CSG, if there is no node with degrees less than 4, we report that there is a *4-core* and the algorithm terminates. Otherwise, we push all the nodes with degrees less than 4 onto the stack and delete them from the CSG. After that, we return to Step 2.
5) We pop all nodes from the stack and color them one by one. The algorithm ends.

Proof of correctness: Here, we prove that if the algorithm reaches the 5^{th} step, the graph can be colored correctly. If coloring a node n_0 leads to conflicts in the the 5^{th} step, there must be more than 4 neighbors of n_0 already colored. However, when n_0 is pushed onto the stack, it has less than 4 neighbors remaining in the graph. Therefore, when n_0 is popped, it also has less than 4 neighbors. As a result, we can safely draw the conclusion that all nodes can be colored successfully without conflicts.

Complexity Analysis: In our RDG algorithm, each node enters the stack at most once. The time complexity of processing each node is related to the number of edges the node has. Each edge is connected to two nodes and is therefore processed at most twice. Therefore, the overall time complexity of the construction is $O(n+m)$. According to Fig 9, our algorithm has high possibility to color the graph successfully with at least 2.2 bits per element on average. Thus, the failure appears at really low rate and hardly increases the time complexity. We have to store all nodes and edges, along with a stack with at most n elements for k-core decomposition, so the space complexity is $O(n+m)$.

4) RDG Update algorithm: Updating refers to inserting an element into or deleting an element from $\mathcal{S}^+$ or $\mathcal{S}^-$. To update the coloring embedder, we propose a method called *RDN (Recursively Delete Neighbor)*: When a node n_i needs to change its color, if there is no candidate color for it, we involve all its neighbors into the modification. The node and its neighbors make up a subgraph. We attempt to color that subgraph using the RDG algorithm recursively. If the subgraph cannot be expanded and cannot be colored, a 4-core is found and the RDG updating algorithm fails.

C. Coloring Embedders for More Than Two Sets

To classify more than two sets, we propose two solutions: The first is to apply a coding method and a fast-memory-access scheme to organize multiple coloring embedders together; the second is to use one large coloring embedder associated with multiple groups of hash functions, which are generated by shifting an original group of hash functions.

1) Coded Representation of Sets: A coded coloring embedder is implemented by multiple coloring embedders. Suppose there are s sets, with IDs ranging from 1 to s. The IDs can be converted to binary codes, with maximum length $\log\lceil s\rceil$. To record the membership of an element, we can record each bit of the set ID binary code with a coloring embedder. This task can be handled by totally $\log\lceil s\rceil$ coloring embedders. If the i^{th} bit is 1, the i^{th} coloring embedder records the element with a positive edge. Otherwise, the coloring embedder records the element with a negative edge. The $\log\lceil s\rceil$ coloring embedders are together called the coded coloring embedder.

We use a fast-memory-access technique to optimize the query speed of the coded coloring embedder. In the above implementation, the number of memory accesses of a coded coloring embedder is $\log\lceil s\rceil$ times that of a single coloring embedder, which slows down the query speed, insertion, and deletion. To address this problem, we reorganize the layout of the $\log\lceil s\rceil$ embedders. All embedders are separated into single bits and the corresponding bits are grouped together. The i^{th} bits of each embedder are grouped into a word, so the binary code of one element can be fetched with only two memory accesses. By using this technique, the coded coloring embedder can work almost as fast as a single coloring embedder.

2) The Shifting Coloring Embedder: The above coded coloring classifier with fast-memory-access can represent more than 2 sets and reduce the number of memory access to 2. However, that approach uses many coloring classifiers and they suffer from a potential load-balancing problem. To address this issue, we propose the Shifting Coloring Embedder, which shares ideas with Shifting Bloom filters [45].

Given s sets with set ID: 0, 1, 2, ..., $s-1$, we build one shifting coloring embedder. There is only one graph and $\log_2 s$ edges are inserted into the graph for each element. We use an example in Figure 6 to show how the shifting coloring embedder works. In this example, there are 8 sets, which

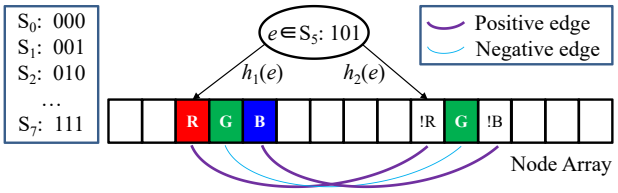


Fig. 6: Shifting coloring embedder.

means $s = 8$ and $\log_2 s = 3$. We assign a code for each set: the code of S_i is i in the binary format. For example, the code of S_5 is 101. When inserting an element e which belongs to set S_5, we compute $h_1(e)$ and $h_2(e)$, and locate $2\log_2 s = 6$ buckets: $n_{h_1(e)}$, $n_{h_1(e)+1}$, $n_{h_1(e)+2}$, and $n_{h_2(e)}$, $n_{h_2(e)+1}$, $n_{h_2(e)+2}$. Since $e \in S_5$ and the code of S_5 is 101 and the first bit of that code is 1 (which corresponds to a positive edge), we create a positive edge between $n_{h_1(e)}$ and $n_{h_2(e)}$. This edge implies that the colors of $n_{h_1(e)}$ and $n_{h_2(e)}$ need to be different. Since the second bit of the code is 0, we create a negative edge and the colors of $n_{h_1(e)+1}$ and $n_{h_2(e)+1}$ need to be the same. Accordingly, the colors of $n_{h_1(e)+2}$ and $n_{h_2(e)+2}$ need to be different.

When $\log_2 s$ is smaller than the length of a machine word, we can answer multi-set query with only two memory accesses. Also, there is no load-balancing problem because there is only one data structure to hold all elements.

IV. Coloring Embedder for Web Cache Sharing

We apply our algorithm to distributed caching at web scale. We first restate the problem of distributed caching to focus on web caches and explore the redundancy generated by the Internet Cache Protocol (ICP) protocol. We discuss the main idea and limitation of prior art, the Summary Cache [22]. We then compare our approach to the performance of Summary Cache.

A. Distributed Caching

Caching is used to reduce Internet bandwidth consumption and to provide faster access speed to content. Moreover, using multiple caches among web proxies on the same side of a link has been shown to decrease network traffic effectively. The Harvest project [11] first proposed the Internet Cache Protocol, which allows a proxy to send queries to neighbors in search of documents in their caches when local cache misses occur.

When a cache miss occurs at a proxy, ICP multicasts a query message to all neighbor proxies in order to probe for potential cache hits. Multicasting ensures a cache hit if the document is cached by at least one of the neighbors. However, multicast also incurs an overhead that increases with the number of proxies. To reduce this overhead, there are two types of queries that we can aim to avoid: queries to proxies that do not cache the document (*invalid queries*) and queries that return duplicate documents from multiple proxies (*unnecessary queries*).

B. Web Cache Sharing with Summary Cache

To address the problem of distributed caching, the most well-known related work is Summary Cache [22], which reduces the query number by reducing invalid queries. The main idea of Summary Cache is shown in Figure 7. Each proxy maintains a compact summary each other proxy that tracks the state of caches at the other nodes. In Summary Cache, the summary consists of multiple on-chip Bloom filters to support quick responses to queries and off-chip Counting Bloom filters to support updates. For each document that incurs a cache miss, a proxy searches all the on-chip Bloom filters and sends a query to proxy i if the query on the i^{th} Bloom filter yields a hit.

Because of the properties of Bloom filters, the Summary Cache has no false negatives and low false positive rate with comparably low memory overhead. If a document is cached at a neighbor proxy, the summary always report the identifier of that neighbor proxy. When the summary reports an identifier of a proxy, it is probable that the proxy caches that document.

We describe the Summary Cache algorithm formally in order to contrast its operation to web cache sharing based on Coloring Embedder in the following subsection.

Summary Cache Algorithm: We number the n proxies from 0 to $n-1$ and regard (hashes of) document names as elements. In each proxy, the summary table consists of $n-1$ Bloom filters, which indicate the elements in other sets. In addition, $n-1$ Counting Bloom filters attached to the Bloom filters are maintained to support delete operation. The Bloom filters are stored in on-chip memory to keep query speed fast and the Counting Bloom filters are stored in off-chip memory because the update speed is less critical in this scenario.

Insertion: When an element e is cached at the i^{th} proxy, the other proxies insert e to the i^{th} Bloom filter and Counting Bloom filter as their inserting operations.

Query: For a query for element e, the proxy searches all $n-1$ Bloom filters and returns a set containing the hits. Identifier i belongs to the set only if the query function of e returns true on the i^{th} Bloom filter.

Deletion: When an element e is removed from the cache, the other proxies first delete e in their Counting Bloom filters. If any counter decreases to 0, then the associated position of the Bloom filter is also set to 0.

The Summary Cache can only reduce invalid queries but not unnecessary queries. If the Summary Cache randomly selects one proxy from the summary set and only queries this proxy in search of the document, then cache miss may occur due to the false positive rate of the Bloom filter. This false positive rate increases linearly with the number of proxies. Thus, the Summary Cache queries all proxies in the summary set (rather than querying only one). As a result, unnecessary queries cannot be avoided and invalid queries may occur occasionally.

C. Web Cache Sharing Based on Coloring Embedder

The Summary Cache has reduced the invalid queries to a low level. Therefore, the unnecessary queries need more concern. We need to cut down unnecessary queries and keep the invalid queries at low level at the same time. Our key idea

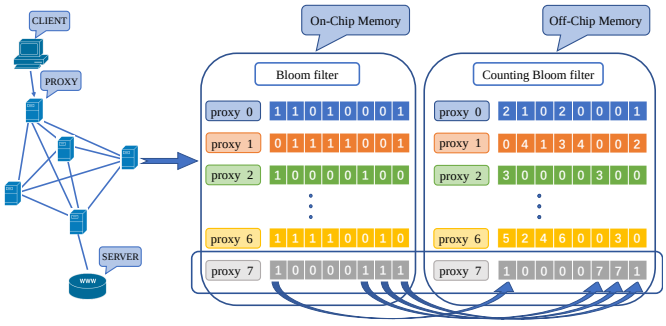


Fig. 7: Example of Summary Cache.

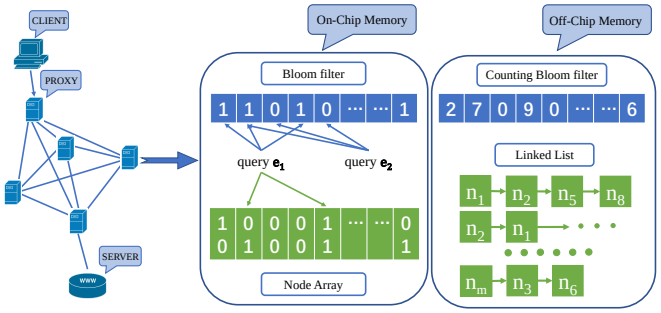


Fig. 8: Example of Coloring Embedder.

is that when a cache miss occurs, this proxy queries only one neighbor proxy and that proxy caches the required document almost every time. In this way, the invalid or unnecessary queries can be reduced to the theoretical minimum.

Though multiple copies may be cached at different proxies of one document, reporting one of them is reasonable and feasible. Considering one miss for one query, we regard this situation as a variant of multi-set query problem and apply Coloring Embedder algorithm for more than two sets (In Section III-C) to the scenario. Differently from the typical multi-set query problem, an element may have several IDs and any one of them is correct under this circumstance.

The main idea of our solution is shown as Figure 8. We perform a similar idea as the Summary Cache on our algorithm. It has two on-chip sections: a large Bloom filter and a node array derived from the Coloring Embedder of the shifting version, and two off-chip sections: a large Counting Bloom filter and a linked list corresponding to them. Differently from the Summary Cache, our Bloom filter records whether a document is cached at any proxy of the proxy group. And we only access the node array to attain a certain result when the Bloom filter indicates that the document may be cached among the proxies.

At the same time, the size of cache is constant, then the number of documents in the cache is within limits, so the Coloring Embedder will never suffer from being overwhelmed if we initialize it with a proper size. The Coloring Embedder can only record one ID for each element, so we record a random ID that is correct.

Query: For an incoming element e, the proxy first queries the Bloom filter to ensure that the element is cached, then computes $h_1(e), h_2(e)$ and locates in the node array, next reads successive $\log\lceil s \rceil$ buckets, finally returns the target ID according to the comparison results.

We take querying element e_1 and e_2 as an example: Assuming there are eight proxies, when querying e_1, the proxy first queries the Bloom filter and returns true. Then it reads two groups of three successive buckets, $(01,\ 00,\ 00)$ and $(11,\ 10,\ 00)$. Therefore, the ID of the target proxy is $110_{(2)} = 6_{(10)}$, and this proxy will send a query to the sixth proxy for element e_1. When querying e_2, the Bloom filter returns false. It indicates that other proxies do not cache e_2 either, and this proxy will send a query to remote server directly.

Insertion and Deletion: We maintain an array to record update items and perform updates when it reaches the threshold. If the cache is not full, then updates only consist of inserting new items, we just perform insertion of the Coloring Embedder. If updates consist of deleting old items and inserting new ones, we first delete some adjacency relations in the linked list and the graph, then we add new edges into this structure, finally we use RDG algorithm to color the graph and write new result to the node array.

By querying in the Bloom filter, our algorithm can also reduce invalid queries to a low level. At the same time, the Coloring Embedder always gives only one but precise ID, so the unnecessary queries hardly take place. As a result, out algorithm creates fewer links and saves the network bandwidth.

Though our algorithm computes hash function two more times than the Summary Cache, our algorithm makes fewer calls to fetch memory. For an incoming element e, our algorithm only searches the Bloom filter once, then computes the $h_1(e), h_2(e)$ and reads successive $\log\lceil n \rceil$ buckets, but the Summary Cache searches all $n-1$ Bloom filters, which means when querying an element, our algorithm can reduce calls to fetch memory. In addition, the number of momery access times increases for the Summary Cache as the number of proxies increases, while it is a constant in our algorithm.

V. Analysis

Two types of errors can occur in our algorithm, which are collision error and color error. Next, we will calculate the expectation of the number of collision errors and the probability that no collision error happens. Then, we will analyze the condition that no coloring error happens. We suppose that there are n buckets in the node array of our data structure, m^+ elements in $\mathcal{S}^+$, and m^- elements in $\mathcal{S}^-$.

A. Collision error

When two edges of different types overlap, a collision error happens. The formal definition is as follow.

Collision error: Given a graph G, a negative connected component is defined as two or more nodes connected by only negative edges. For each negative connected component N^-, if there are two nodes $n_1, n_2 \in N^-$ which are directly connected by a positive edge, a collision error happens.

1) Simple Cases:

The analysis begins with the simplest case of collision error: a positive edge overlaps with a negative edge. To discuss the worst case, we suppose that there are m^+ non-overlapped positive edges and m^- non-overlapped negative edges in the graph. The probability that a negative edge collides with any positive edge is $m^+/\binom{n}{2} = 2m^+/[n(n-1)]$. To calculate the upper bound [28]–[30] of the expectation of the number of collision errors, we can directly sum up the probability of collisions for all negative edges because they follow a binomial distribution.

$$E(collision) \leqslant \frac{m^+m^-}{\binom{n}{2}} = \frac{2m^+m^-}{n(n-1)} \tag{1}$$

Given a negative edge, the probability that it does not collide with any positive edge is $1 - m^+/\binom{n}{2}$. Collisions are independent events for each negative edge because we suppose they do not overlap with each other, so we can apply the multiplication principle to get the lower bound of the probability that there is no collision error.

$$\begin{aligned} P(no\ collision) &\geqslant \left(1 - \frac{m^+}{\binom{n}{2}}\right)^{m^-} \approx \left(1 - \frac{2m^+}{n^2}\right)^{m^-} \\ &= \left(1 - \frac{2m^+}{n^2}\right)^{\frac{n^2}{2m^+} \times \frac{2m^+}{n^2} m^-} \approx \mathbf{e}^{-\frac{2m^+m^-}{n^2}} \end{aligned} \tag{2}$$

2) General Cases:

In this section, we analyze a more complex situation of collision error: two nodes are indirectly connected by a list of continuous negative edges, and are at the same time directly connected by a positive edge. For convenience, we name the list of continuous negative edges as an *equivalent negative edge*. To calculate the expectation of the number of collision error, we need to count the number of equivalent negative edges, which is denoted as m'^-.

Our analysis begins with deriving the number of equivalent negative edges formed by two negative edges between three nodes. Given three nodes, the probability that two of them are directly connected by a negative edge is $\frac{3}{n} \times \frac{2}{n} m^-$. And the probability that another pair of nodes is also directly connected by a negative edge is $\frac{2}{n} \times \frac{1}{n}(m^- - 1)$. So the probability that three nodes are connected by two negative edges is

$$\frac{12m^-(m^- - 1)}{n^4}$$

For all the n nodes, the expectation of the number of equivalent negative edges formed by three nodes is then calculated by the following equation.

$$m'^-_{(3)} = \frac{12(m^-)^2}{n^4}\binom{n}{3} \approx \frac{2(m^-)^2}{n} \tag{3}$$

The number of equivalent negative edges formed by four or more nodes can be similarly derived. Given any v nodes, the probability that they are connected by $v-1$ negative edges is

$$\frac{v!}{2}\prod_{i=1}^{v-1}\frac{m^- - i + 1}{\binom{n}{2}} < \frac{2^{v-2}(m^-)^{v-1}}{n^{v-1}(n-1)^{v-1}}$$

The number of equivalent negative edges formed by v nodes is the product of that value and $\binom{n}{v}$.

$$m'^-_{(v)} < \frac{2^{v-2}(m^-)^{v-1}}{n^{v-1}(n-1)^{v-1}}\binom{n}{v} < \frac{2^{v-2}n(m^-)^{v-1}}{(n-1)^{v-1}} \tag{4}$$

From equation 4, we can find that the number of equivalent negative edges is approximate to a geometric progression when the value of v increases. We only show the case that n is larger than $2m^-$. Other cases can be deduced similarly.

$$\begin{aligned} m'^- &= \sum_{i=3}^{min(n,\ m^- - 1)} m'^-_{(i)} \\ &< \sum_{i=3}^{min(n,\ m^- - 1)} \frac{2^{i-2}n(m^-)^{i-1}}{(n-1)^{i-1}} \\ &< \frac{2(m^-)^2 n}{(n-1)(n-2m^- - 1)} \\ &\approx \frac{2(m^-)^2}{n - 2m^-}, \quad n > 2m^- \end{aligned} \tag{5}$$

To get the final result of the expectation of the number of collisions and the probability that no collision happens, the m^- in the simplified equation 1 and 2 is replaced by $m^- + m'^-$. n is required to be larger than $2m^-$ in practice. The reason is that if n is smaller than $2m^-$, the graph can hardly be colored successfully.

$$\begin{aligned} E(collision) &\leqslant \frac{2m^+(m^- + m'^-)}{n(n-1)} \\ &\approx \frac{2m^+m^-}{n(n-2m^-)} \end{aligned} \tag{6}$$

$$\begin{aligned} P(no\ collision) &\geqslant \mathbf{e}^{-\frac{2}{n(n-1)}m^+(m^- + m'^-)} \\ &\approx \mathbf{e}^{-\frac{2m^+m^-}{n(n-2m^-)}} \end{aligned} \tag{7}$$

Let `n/m ratio` be the quotient of n divided by m. According to equation 6 and 7, the expectation of the number of collision errors and the probability that no collision error happens are not influenced by the graph size when `n/m ratio` is fixed. When `n/m ratio` is larger than 1.1, which means each element uses more than $1.1 \times 2 = 2.2$ bits, the expectation of the number of collision errors is less than 5 no matter how many elements there are, and the probability that no collision error happens is larger than 50%.

B. Color error

When the graph is dense, 4 colors may be not enough to make all edges in the graph meet the coloring rule. For example, suppose there are 5 nodes in the graph and each pair of nodes is connected by a positive edge, then 5 different colors are required to make all pairs of nodes have different colors to meet the coloring rule. The formal definition of color error is as follows.

Color error: The graph cannot be colored successfully with four colors by the RDG coloring algorithm.

In our RDG coloring algorithm, we give up coloring if we find a 4-core in the graph. As a result, color error happens when there is a 4-core. Theories about k-cores in random graphs are established in [35].

1) If $k \geqslant 3$ and n is large, with high probability, there is a giant k-core when m_+ is larger than $c_k n/2$ and there is no k-core when m_+ is smaller than $c_k n/2$.

2) $c_k = k + \sqrt{k \log n_0} + O(\log n_0)$.

According to [35], c_4 is calculated to be *5.14*. The n/m *ratio* threshold for color error is equal to $2/c_4$. Therefore, when there is no negative edge in our graph, the n/m *ratio* threshold is *0.389*. When there are negative edges, our graph is not a random graph and thus the results in [35] do not apply. From the perspective of coloring, negative edges combine many nodes into a large single node because those nodes must have the same color. The large single node has many neighbors, and thus the subgraph containing that node can be very dense, leading to a higher probability of the emergence of a 4-core. As a result, n/m *ratio* threshold becomes larger when the percentage of negative edges is higher. In the worst case, when the negative edges account for 50% of all edges, the n/m *ratio* threshold is *1.10* according to our experiments. In conclusion, we need no more than $1.10 \times 2 =$*2.20* bits per element to build a coloring embedder to ensure that no color error happens.

3) The upper bound of probability of the 4-core.

We consider the probability of a 4-core when the nodes that are connected by negative edges have been merged as one. Given a subgraph with n_0 nodes and n_1 negative edges before merging, suppose there are n_2 nodes after merging. Obviously, we have $n_2 \geqslant 4$. If the new subgraph is a 4-core, there are at least $2n_2$ positive edges in it and n_1 negative edges in the subgraph of these n_0 nodes, where $n_0 = n_1 + n_2$. The number of edges in this subgraph is at least

$$2 \cdot n_1 + n_2 \geqslant 4 + n_1 + n_2 = n_0 + 4 \tag{8}$$

So the probability of a 4-core which has n_0 nodes in the original graph P_{n_0} satisfies

$$\begin{aligned} P_{n_0} &\leqslant \binom{n}{n_0} \cdot \binom{m}{n_0+4} \cdot \left(\frac{\binom{n_0}{2}}{\binom{n}{2}} \right)^{n_0+4} \\ &< \frac{n^{n_0}}{n_0!} \cdot \frac{m^{n_0+4}}{(n_0+4)!} \cdot \frac{n_0^{2n_0+8}}{n^{2n_0+8}} \\ &< \frac{m^{n_0} \cdot n_0^{2n_0+8}}{(n_0!)^2 \cdot n^{n_0+8}} \\ &= \frac{m^{n_0} \cdot n_0^{2n_0+8}}{\left(\sqrt{2\pi n_0} \cdot \left(\frac{n_0}{e}\right)^{n_0}\right)^2 \cdot n^{n_0+8}} \\ &= \frac{e^{2n_0} \cdot m^{n_0} \cdot n_0^7}{2\pi \cdot n^{n_0+8}} \\ &< \frac{1}{2\pi n e^{14}} \cdot \left(\frac{e^2 m}{n}\right)^{n_0+7} \end{aligned} \tag{9}$$

When $\frac{n}{m} \geqslant \frac{e^2}{0.99}$, the probability of 4-core P_{4-core} satisfies

$$\begin{aligned} P_{4-core} &\leqslant \sum_{n_0=5}^{n} P_{n_0} < \frac{1}{2\pi n e^{14}} \cdot \sum_{n_0=5}^{\infty} 0.99^{n_0} \\ &< \frac{1}{2\pi n e^{14}} \cdot \frac{1}{1-0.99} < \frac{1}{10^6 n} \end{aligned} \tag{10}$$

VI. Experimental Results

A. Experimental Setup

1) Datasets:

We use three real datasets and generate plenty of synthetic datasets for experiments. The statistics of the real datasets are shown in Table III.

MACTable: This dataset is drawn from the MAC table file in [2]. For each entry in the MAC table, we use the line number as the key, and use the type field (static or dynamic) to determine the set.

MachineLearning: This dataset is a dataset of binary classification task from UCI machine learning repository [4]. We use the training set to evaluate the performance of our algorithm. For each entry in the training set, we use the line number as the key, and the label as the class.

DBLP: This dataset is drawn from DBLP [1]. We use the `key` attribute as our key. We use the records of articles as $\mathcal{S}^+$ and the records of inproceedings as $\mathcal{S}^-$.

TABLE III: Statistics of the real datasets.

	# items	m^+	m^-	$\mathcal{S}^-$ ratio
MACTable	3664	3144	520	0.1419
MachineLearning	912969	472605	440364	0.4823
DBLP	823132	623212	199920	0.2429

Synthetic dataset: We generate random strings as keys of elements in a dataset. A specific number of random unique integers are taken from uniform distribution and divided into two sets. Therefore, the ratio of S^- can be adjusted by changing the size of the corresponding set. We use synthetic datasets because our data structure have to be examined when the percentage of $\mathcal{S}^-$ is continuously changing, while real world datasets have fixed percentage of $\mathcal{S}^-$. We argue that for data structures using hash functions, including the coloring embedder, real datasets and synthetic datasets have no difference. The experiments in the next section also prove this fact. Therefore, we use the synthetic dataset for the experiments on more than two sets (See in Section VI-C). We can divide the random strings into 4, 8 or 16 sets for multi-set experiments.

Datasets for web cache sharing: Since we are unable to collect all original datasets used in Summary Cache as they are too old, we use two datasets that are from the University of California at Berkeley Dial-IP service [24], same as the UCB dataset used in Summary Cache. We see HTTP requests in both datasets as requests for certain documents from certain users to perform simulation. For convenience, we call the two datasets as UCB1 and UCB2. Table IV lists the details of the two datasets. We add the size of all unique documents of both datasets as the total size in the table.

TABLE IV: Description of the datasets.

Name	Requests #	Documents #	Total size
UCB1	2408297	819809	6.4GB
UCB2	1569105	592224	4.58GB

2) The State-of-the-art Implementation:

To compare our data structure with the state-of-the-art, we implement three Bloom filter based data structures used for multi-set query. The first one is the Multiple Bloom filter [48]. The Multiple Bloom filter simply assembles the Bloom filters, each one representing one of the sets. This model is called MultiBF in short. The second data structure is the Coded Bloom filter [13], denoted as CodedBF. It is a typical variance of Bloom filter using multiple filters. It converts set IDs to binary codes and stores the code in the Bloom filters. The third one is the shifting Bloom filter [45], denoted as ShiftBF. It is a typical variance of Bloom filter using a single filter. ShiftBF uses the offset of bits to represent the set ID.

We also implement two data structures that need to store the key or its fingerprint: a multi-way cuckoo filter [21] that assembles cuckoo filters for each set, and a key-value hash table with seperate chaining and load factor 1.

The source code of coloring embedder is released on Github [3].

3) Running Environment:

We use general-purposed CPU to run all experiments, because we do not have FPGA or ASIC environment. We conduct all experiments on a standard off-the-shelf computer equipped with two 18-core Intel(R) Core i9-10980XE CPUs @3.00GHz and 128GB RAM running Ubuntu 20.04. For each core, the L1 data cache is 32KB and the L2 cache is 1MB.

B. Experiments on Two Sets

In this section, we conduct experiments on two-set query, which is the foundation of multi-set query. We use real datasets and synthetic datasets to comprehensively evaluate performance of hyper mapping and coloring embedding, and measure the throughput of construction, query and insertion.

1) Coloring Embedding:

First, we show that there is a sharp threshold for successful coloring embedding. Then, we test the condition of successful coloring embedding when the percentage of $\mathcal{S}^-$ varies.

Successful coloring rate vs. n/m ratio (Figure 9): *The experimental results show that there is a sharp threshold of n/m ratio for the success rate of coloring embedding.* In this experiment, we test the success rate against the n/m ratio on all three real datasets. For each real dataset, with the same percentage of $\mathcal{S}^-$, we generate synthetic datasets of different sizes, varies from 10^3 to 10^6. The results are shown in Figure 9. As the *n/m ratio* increases, there is an almost-zero success rate when the *n/m ratio* is below the threshold, a similar surge when the *n/m ratio* is passing the threshold, and an almost-one success rate when the *n/m ratio* is above the threshold. The thresholds are 0.52, 1.07, and 0.66 for datasets of MACTable, MachineLearning, and DBLP, respectively. The threshold of synthetic datasets is the same with that of real datasets [41], [42], [53]. The larger the datasets are, the sharper the threshold is. The threshold for different real datasets are different, because the percentage of $\mathcal{S}^-$ is different. MACTable dataset has the smallest *n/m ratio* threshold because it has the smallest percentage of $\mathcal{S}^-$. There is no sharp threshold for small datasets in Figure 9(b), because they have different properties from large datasets.

Memory needed vs. Percentage of $\mathcal{S}^-$ (Figure 10): *The experimental results show that the memory needed for coloring embedding increases when the percentage of $\mathcal{S}^-$ increases.* We measure the memory needed (bits per element) in the condition that the successful coloring rate is above 99%. The three real datasets are displayed as points in the figure, while the synthetic datasets are displayed as a line. When the percentages of $\mathcal{S}^-$ is around 13%, the memory needed is below *1* bit per element.

When the percentages of $\mathcal{S}^-$ is around 50%, which is the worst case of our algorithm, the memory needed is *2.2m* bits, where m is the number of elements. When the memory size is larger than 2.2 bits per element, the graph is sparse enough so that there is no 4-core and thus can be colored successfully with 4 colors. *2.2* bits per element is always enough for all kinds of datasets because when the percentages of $\mathcal{S}^-$ is larger than 50%, we can simply exchange $\mathcal{S}^-$ with $\mathcal{S}^+$.

2) Hyper Mapping:

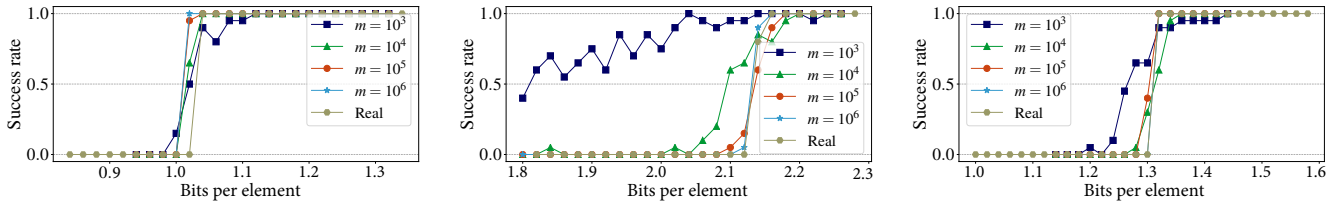
In this part, we evaluate the number and probability of edge collisions during hyper mapping under different settings of n/m *ratio* and the percentage of $\mathcal{S}^-$. We use synthetic datasets with sizes from 10^3 to 10^6. By default, the percentage of $\mathcal{S}^-$ is 50%, and the n/m *ratio* is *1.1*, which is the threshold of successful coloring.

Number of collisions vs. n/m ratio (Figure 11(a)): *The experimental results show that the average number of collisions decreases when the n/m ratio increases.* Specifically, when the n/m *ratio* is *1.4*, the expectations of the number of collisions of all datasets are 1. The number of collisions is not influenced by dataset sizes when the n/m *ratio* is above *1.15*. The experimental results fit well with the theory.

Probability of collisions vs. n/m ratio (Figure 11(b)): *The experimental results show that the probability that collisions happen decreases when n/m ratio increases.* The probability that collisions happen is not influenced by the set size. When the n/m *ratio* is *1.3*, the probability that collision happens is about 75%. And when the n/m *ratio* is *1.5*, the probability is 50%. The experimental results fit well with the theory.

Number of collisions vs. Percentage of $\mathcal{S}^+$ (Figure 11(c)): *The experimental results show that the number of collisions decreases when percentage of $\mathcal{S}^+$ increases.* In this experiment, we change the percentage of $\mathcal{S}^+$ from 45% to 100%, and fix the n/m *ratio* to *1.1*. From Figure 11(c) we can see that when $\mathcal{S}^+$ accounts for more than 50%, there are less than 2 collisions for datasets of all sizes. When there is no negative edge, there is no collision. The experimental results fit well with the theory.

Probability of collisions vs. Percentage of $\mathcal{S}^+$ (Figure 11(d)): *The experimental results show that the probability*



(a) MACTable (b) MachineLearning (c) DBLP

Fig. 9: Successful coloring rate vs. n/m *ratio.*

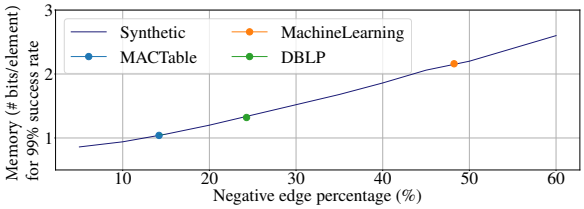


Fig. 10: Memory needed vs. percentage of $\mathcal{S}^-$.

that collisions happen decreases when percentage of $\mathcal{S}^+$ increases. The probability is almost not influenced by the size of datasets. It decreases almost linearly from about 90% to 0% when percentage of $\mathcal{S}^+$ increases from 50% to 100%. The experimental results fit well with the theory.

Throughput of insertion vs. Dataset size (Figure 12): *The experimental results show that the throughput of insertion is high when the load rate is below a threshold.* In this experiment we first construct an empty coloring embedder using 2.21×10^6 bits, and then insert 10^6 elements into it. Figure 12(a) shows that when we insert less than 65% elements into the coloring embedder, few nodes are affected by the RDG updating algorithm. In contrast, when we insert more than 65% elements, tens of thousand nodes need to be recolored. Figure 12(b) shows that the insertion speed decreases gradually when we insert less than 65% elements, and drops sharply when we insert more than 65% elements. The insertion speed is still above 0.4 MOPS when 65% elements are already in the coloring embedder. This means our algorithm supports fast insertion when the load rate is less than 65% and promises that it will not involve into a loop.

C. Experiments on Multi-sets

In this section, we compare shifting coloring embedder with Bloom filters, cuckoo filters and hash table on multi-set query. We use synthesis datasets to test the worst case of our coloring embedder (each sets has the same size). First we fix the number of sets to 16 and vary the dataset size from 10^3 to 10^6. Then we fix the set size to 10^6 and vary the number of sets from 2 to 16. The sizes of data structures are adjusted to assure that the number of errors is limited under 10 in each experiment.

Throughput of query vs. Dataset size (Figure 13(a)): *The experimental results show that our shifting coloring embedder has faster query speed compared with other approaches.* The number of sets is fixed to 16 while the size changes. The query speed of the shifting coloring embedder is around 60 MOPS when there are 10^3 elements. And its query speed drops to 40 MOPS when the number of elements increases to 10^6. The query speed of Bloom filters is always less than 20 MOPS. The query speed of cuckoo filter ranges from 14.4 MOPS to 30.1 MOPs. The query speed of hash table is as fast as color embedder when the dataset is small, but its speed drops to under 20 MOPS when the dataset is large as 10^6.

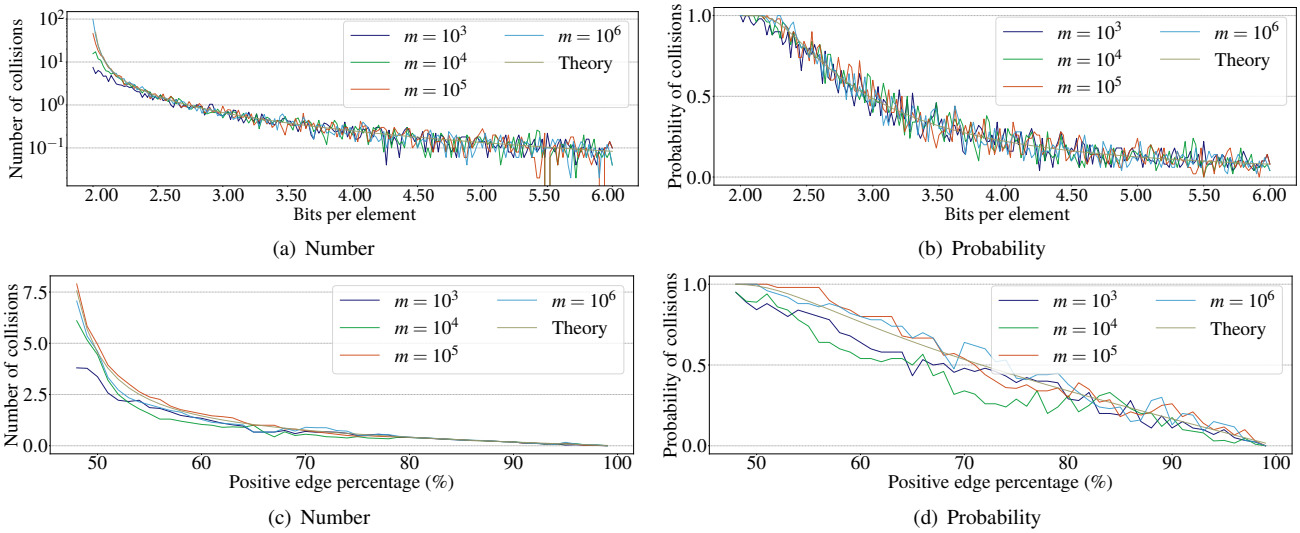
Memory vs. dataset size (Figure 13(b)): *The experimental results show that the shifting coloring embedder uses the least memory for different dataset sizes.* The number of sets is fixed to 16 and the number of errors is limited under 10. Bloom filters use more than 15 bits per element, and use more when size of dataset increases, while color embedder uses 8.9 bits per element to implement high accuracy in queries. Cuckoo filter uses more memory than MultiBF and ShiftBF per element because the fingerprint of keys need to be stored. And hash table with separate chaining use much more bits because pointers are also stored besides keys.

Throughput of query vs. Number of sets (Figure 13(c)): *The experimental results show that the shifting coloring embedder has the fastest query speed for different number of sets.* We change the number of sets from 2 to 16 and test query speed on 10^6 elements. The query speed of color embedder varies from 96.2 MOPS to 41.0 MOPS. And the query speed of all other data structures is lower than 20 MOPS.

Memory vs. Number of sets (Figure 13(d)): *The experimental results show that the shifting coloring embedder uses the least memory when varying the number of sets.* The dataset size is fixed to 10^6 and number of errors is limited under 10. When varying the number of sets from 2 to 16, CodedBF, MultiBF, ShiftBF and our algorithm uses 13 to 56, 24 to 31, 24 to 31 and 2.2 to 8.9 bits per element memory, respectively. Cuckoo filter uses more memory than MultiBF and ShiftBF, and hash table uses 200 bits per element, because the two data structures need to store extra information of keys, while bloom filters and color embedder just use bit arrays.

D. Experiments on web cache simulation

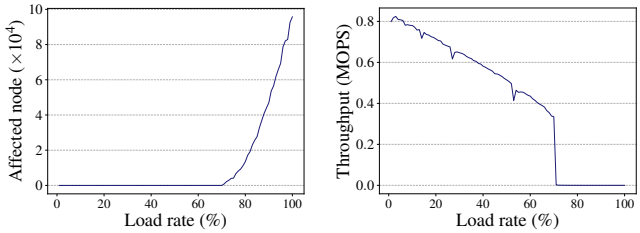
1) Preprocessing: For both datasets, each item indicates a request for a document from a client. We partition the clients into several groups using hash functions according to their IP addresses. We set the same number of proxies for different methods, which is 8 by default, assuming that a request from a client partitioned into a certain group will go into the corresponding proxy of the group. We set the cache size of each proxy as 10% of the total size and apply the least



(a) Number (b) Probability

(c) Number (d) Probability

Fig. 11: Number and probability of edge collisions.



(a) number of affected nodes (b) throughput

Fig. 12: Insertion speed vs. Load rate.

recently used (LRU) algorithm to the caches. We filter out those items with documents larger than 250KB in the original records, since documents with such large size are not likely to be cached in practice.

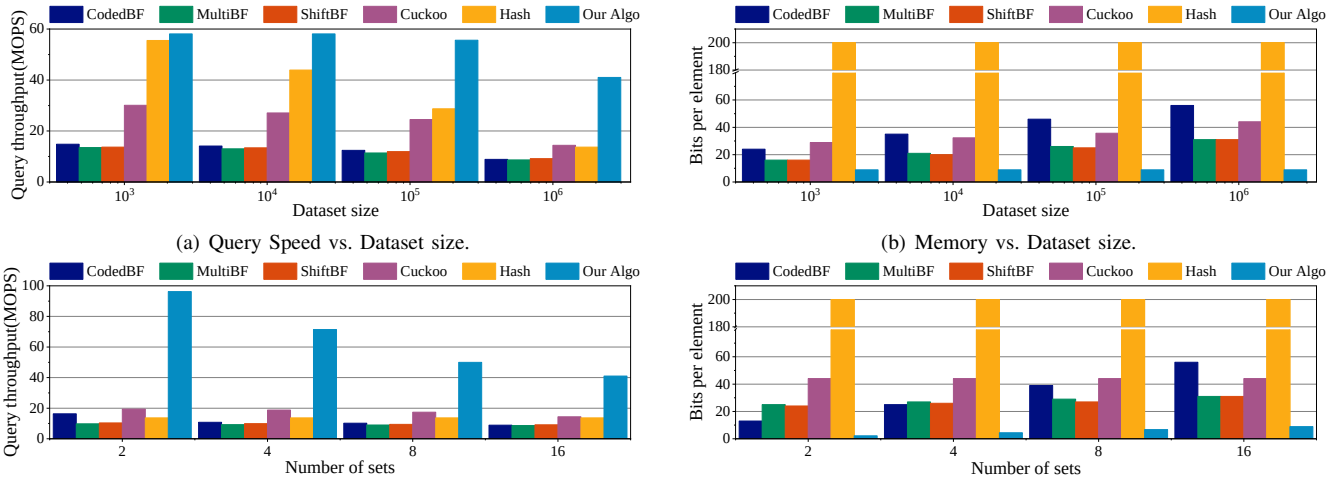
2) Metrics: When a request reaches a proxy, the proxy will first check whether the requested document is stored in its cache. If so, the proxy will return the document at once. Otherwise, the proxy will check its summary, find out candidate proxies, and send inter-proxy queries to the selected proxies and receive returned copies of the document to report to the client. According to the processes, we use two metrics to measure the performance of the summary approaches.

Average Hit Rate: A request is marked as a hit if the proxy returns the correct document without sending requests to remote Web servers, otherwise it is marked as a miss if the proxy cannot return the correct document after requesting among the proxies and finally send requests to Web servers. When a hit takes place, the document can either come from the first local proxy or come from other proxies via inter-proxy queries. The Average Hit rate is defined that the ratio of the number of hit requests divided by the number of all request documents. The Average Hit Rate is the most important metric because a high hit ratio can reduce traffic to Web servers and shorten the response time.

Average Request Rate: When a request from a client reaches a proxy, the proxy may send some inter-proxy queries to other proxies that might store the document indicated by the summary algorithm. Since inter-proxy queries will increase traffic, a low average inter-proxy query number while maintaining the average hit rate on a high level is desired. We define Average Request Rate as the total number of requests sent to other proxies divided by the number of all request documents.

3) Delay Threshold for Updating: During the simulation of the cache querying, cache will be frequently replaced. However, it will waste a lot of time and traffic if we update the summaries of all proxies immediately once one cache has been changed. At the same time, if the summary has not been updated for a long time, it will cause many invalid requests because the documents that summary indicates may have been replaced, and new documents are not updated by the summary either. We define the delay threshold as the proportion of the cache that has been changed before the changes of the cache are broadcast to other proxies and the summaries are updated. The delay threshold is a trade off between accuracy and updating cost, while the bandwidth cost has a strong connection with the accuracy, so it is a balance between bandwidth and updating indeed. In our experiments, we simulate cache updating when the delay threshold is 1%, 2%, 5%, and 10%, just the same as the Summary Cache.

Average Request Rate vs. methods (Figure 14): *The experimental results show that our Coloring Embedder with Bloom filter can decrease the average request rate obviously compare with the Summary Cache.* In this section, we evaluate the average request rate of three methods on both datasets and the number of proxies is 8 and 16. The "Summary Cache" refers to method of multiple Bloom Filters, "Coloring Embedder" refers to our original data structure and "Coloring with BF" refers to the method of first filtering the requests by a Bloom Filter before processing them on Coloring Embedder. In addition, the on-chip memory usage for such three methods is the same for fairness. According to the experimental results, we find that the average request rate of the Summary Cache increases from 0.3 to 0.6 as the number of proxies is doubled, because more proxies will cache redundant copies and the Summary Cache queries all candidate proxies. As for our Coloring Embedder,



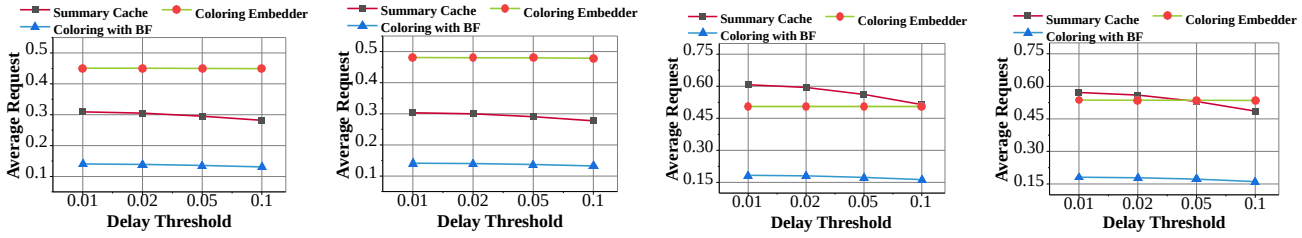
(a) Query Speed vs. Dataset size.

(b) Memory vs. Dataset size.

(c) Query Speed vs. Number of sets.

(d) Memory vs. Number of sets.

Fig. 13: Performance on different dataset sizes and different numbers of sets



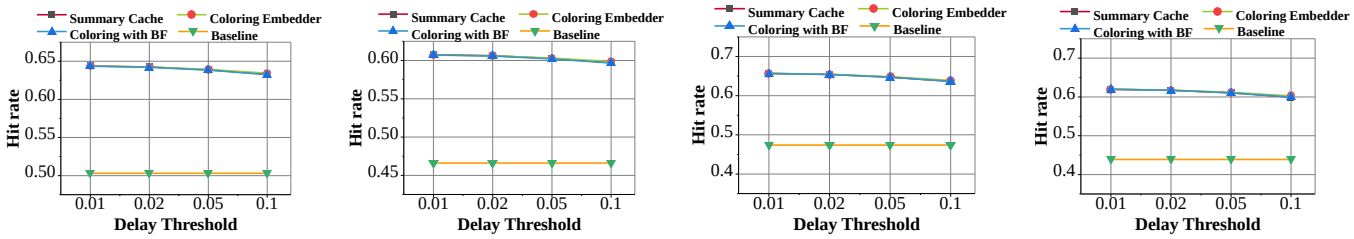
(a) 8proxiesUCB1

(b) 8proxiesUCB2

(c) 16proxiesUCB1

(d) 16proxiesUCB2

Fig. 14: Average request rate of different methods.



(a) 8proxiesUCB1

(b) 8proxiesUCB2

(c) 16proxiesUCB1

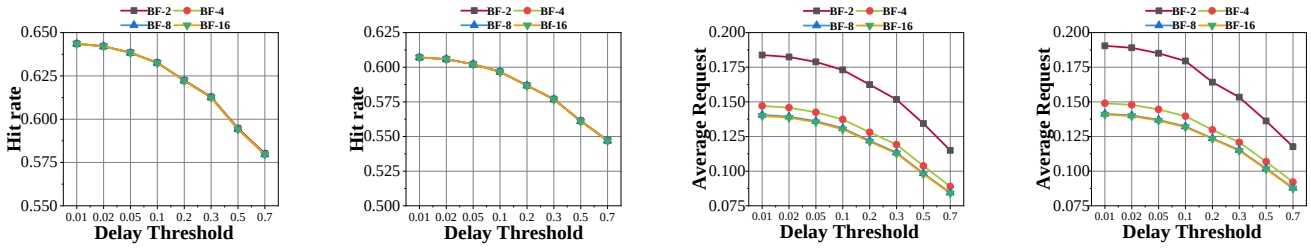
(d) 16proxiesUCB2

Fig. 15: Average hit rate.

the average request rate stays unchanged when scaling up the proxies. In addition, after filtering by the Bloom filter, we can avoid the request generated by obligation miss.

Average Hit Rate vs. methods (Figure 15): *The experimental results show that our Coloring Embedder (with or without Bloom filter) can reach nearly the same hit rate as the Summary Cache.* In this section, we compare the average hit rate of these methods. We also simulate the hit rate that using only local cache at each proxy and sending no queries to other proxies, and we call it "Baseline". The "Baseline" is drawn to state the significance of querying other proxies. From the properties of the Bloom filter, we have known that the Summary Cache reaches the highest hit rate regardless of updating. Also, the slight difference may be caused by the delay threshold or diverse updating sources selected. Considering the above two experiments comprehensively, we maintain the average hit rate and reduce the average request rate by applying our algorithm.

Request or Hit Rate vs. Size of Bloom filter (Figure 16): *The experimental results show that using 8 bits on average is the best choice to save memory and reduce the request rate.* In this section, we tend to find the best size of the Bloom filter that is used for filtering the documents causing obligation misses. We maintain a Bloom filter to judge a document is cached at any one of the proxies, and update it using off-chip Counting Bloom filter. BF-n means that we use n bits on average for every unique document in the dataset. We also increase the threshold to a very large value in order to observe performances of Coloring Embedder with different sizes of Bloom filter. According to the experimental results, we find that using 8 bits on average reaches similarly performance as using 16 bits, and better than using fewer bits. When the threshold keeps increasing to a large value(greater than 0.1), the performance would quickly deteriorate. However, in the actual application scenario, the threshold will not reach such



(a) 8proxiesUCB1 (b) 8proxiesUCB2 (c) 8proxiesUCB1 (d) 8proxiesUCB2

Fig. 16: Different size of Bloom filter.

a high level. The cache is expected to be updated with a small threshold in practice, so as not to affect the query accuracy.

VII. Conclusion

In this paper, we propose a novel data structure named coloring embedder. The coloring embedder is used for two-set query, and a shifting model is designed for the coloring embedder to support multi-set query. Experimental results show that our coloring embedder can achieve less than 5 errors and high success rate of construction on data sets containing 10^7 elements with only $2.2 \log s$ bits per element memory in the worst case, where s is the number of sets. In addition, our coloring embedder achieves lower request rate with nearly the same hit rate compared to the Summary Cache applying to the web cache sharing. The source code of coloring embedder is released on Github [3].

Acknowledgment

Tong Yang is the corresponding author. This work is supported by National Natural Science Foundation of China (NSFC) (No. U20A20179), and the project of "FANet: PCL Future Greater-Bay Area Network Facilities for Large-scale Experiments and Applications" (No. LZC0019).

References

[1] dblp: computer science bibliography. http://dblp.org/xml/release/dblp-2017-09-03.xml.gz.
[2] Hassel library. https://bitbucket.org/peymank/hassel-public.
[3] The source code of coloring embedder. https://github.com/4colorclassifier/4colorclassifier.
[4] Youtube comedy slam preference data data set. https://archive.ics.uci.edu/ml/datasets/YouTube+Comedy+Slam +Preference+Data.
[5] M. K. Aguilera, W. M. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, 2008.
[6] V. Batagelj and M. Zaversnik. An o(m) algorithm for cores decomposition of networks. *Computer Science*, 1(6):34–37, 2003.
[7] D. Belazzougui, F. C. Botelho, and M. Dietzfelbinger. Hash, displace, and compress. *Lecture Notes in Computer Science*, 5757:682–693, 2009.
[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.
[9] B. Bollobás, S. Janson, and O. Riordan. The phase transition in inhomogeneous random graphs. *Random Structures & Algorithms*, 31(1):3–122, 2010.
[10] F. C. Botelho, Y. Kohayakawa, and N. Ziviani. *A Practical Minimal Perfect Hashing Method*. Springer Berlin Heidelberg, 2005.
[11] M. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28:119–125, 12 1995.
[12] W. Bux, W. E. Denzel, T. Engbersen, A. Herkersdorf, and R. P. Luijten. Technologies and building blocks for fast packet forwarding. *IEEE Communications Magazine*, 39(1):70–77, 2001.
[13] F. Chang, W.-c. Feng, and K. Li. Approximate caches for packet classification. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2196–2207. IEEE, 2004.
[14] J. Cheng, Y. Ke, S. Chu, and M. T. Ozsu. Efficient core decomposition in massive networks. In *IEEE International Conference on Data Engineering*, pages 51–62, 2011.
[15] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992.
[16] H. Dai, L. Meng, and A. X. Liu. Finding persistent items in distributed, datasets. In *Proc. IEEE INFOCOM*, 2018.
[17] H. Dai, M. Shahzad, A. X. Liu, and Y. Zhong. Finding persistent items in data streams. *Proceedings of the VLDB Endowment*, 10(4):289–300, 2016.
[18] H. Dai, Y. Zhong, A. X. Liu, W. Wang, and M. Li. Noisy bloom filters for multi-set membership testing. In *Proc. ACM SIGMETRICS*, pages 139–151, 2016.
[19] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. A. D. Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Foundations of Computer Science, 1988., Symposium on*, pages 524–531, 1988.
[20] R. Durrett. *Random graph dynamics*, volume 200. Cambridge university press Cambridge, 2007.
[21] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.
[22] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM ToN*, 8(3):281–293, 2000.
[23] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. W. H. Freeman, 1979.
[24] S. Gribble. Uc berkeley home ip http traces. 01 1997.
[25] F. Hao, M. Kodialam, T. V. Lakshman, and H. Song. Fast dynamic multiple-set membership testing using combinatorial bloom filters. *IEEE/ACM Transactions on Networking*, 20(1):295–304, 2012.
[26] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo. K-core decomposition of large networks on a single pc. *Proceedings of the VLDB Endowment*, 9(1):13–23, 2015.
[27] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Science and Engineering*, pages 1–16, 2021.
[28] Z. Li, B. Chang, S. Wang, A. Liu, F. Zeng, and G. Luo. Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things. *IEEE Transactions on Industrial Informatics*, 2018.
[29] Z. Li, Y. Liu, A. Liu, S. Wang, and H. Liu. Minimizing convergecast time and energy consumption in green internet of things. *IEEE Transactions on Emerging Topics in Computing*, 2018.
[30] Z. Li, F. Xiao, S. Wang, T. Pei, and J. Li. Achievable rate maximization for cognitive hybrid satellite-terrestrial networks with af-relays. *IEEE Journal on Selected Areas in Communications*, 36(2):304–313, 2018.
[31] G. Lu, Y. J. Nam, and D. H. Du. Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.
[32] Y. Lu, B. Prabhakar, and F. Bonomi. Bloom filters: Design innovations and novel applications. (1):201–206, 2005.
[33] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[34] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[35] B. Pittel, J. Spencer, and N. Wormald. Sudden emergence of a giant k -core in a random graph. *Journal of Combinatorial Theory*, 67(1):111–151, 1996.

[36] Y. Qiao, S. Chen, Z. Mo, and M. Yoon. When bloom filters are no longer compact: Multi-set membership lookup for network applications. *IEEE/ACM Transactions on Networking*, 24(6):3326–3339, 2016.

[37] A. E. Sarıyüce, B. Gedik, G. Jacques-Silva, K. L. Wu, and Ümit V. Çatalyürek. Incremental k -core decomposition: algorithms and evaluation. *VLDB Journal*, 25(3):425–447, 2016.

[38] S. Soleymani, M. Anisi, A. H. Abdullah, M. A. Ngadi, S. Goudarzi, M. K. Khan, and M. N. Kama. An authentication and plausibility model for big data analytic under los and nlos conditions in 5g-vanet. *Science China Information Sciences*, 63(12):1–17, 2020.

[39] B. Tran, B. Schaffner, J. M. Myre, J. Sawin, and D. Chiu. Exploring means to enhance the efficiency of gpu bitmap index query processing. *Data Science and Engineering*, pages 1–20, 2020.

[40] S. Wu, D. Jiang, B. C. Ooi, and K.-L. Wu. Efficient b-tree based indexing for cloud data processing. *Proceedings of the VLDB Endowment*, 3(1-2):1207–1218, 2010.

[41] F. Xiao, L. Chen, C. Sha, L. Sun, R. Wang, A. X. Liu, and F. Ahmed. Noise tolerant localization for sensor networks. *IEEE/ACM Transactions on Networking*, 26(4):1701–1714, 2018.

[42] F. Xiao, Z. Wang, N. Ye, R. Wang, and X.-Y. Li. One more tag enables fine-grained rfid localization and tracking. *IEEE/ACM Transactions on Networking (TON)*, 26(1):161–174, 2018.

[43] D. Yang, D. Tian, J. Gong, S. Gao, T. Yang, and X. Li. Difference bloom filter: A probabilistic structure for multi-set membership query. In *IEEE International Conference on Communications*, pages 1–6, 2017.

[44] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig. Elastic sketch: Adaptive and fast network-wide measurements. In *Proc. ACM SIGCOMM 2018*, pages 561–575.

[45] T. Yang, A. X. Liu, M. Shahzad, and et al. A shifting bloom filter framework for set queries. *Proceedings of the VLDB Endowment*, 9(5):408–419, 2016.

[46] T. Yang, G. Xie, Y. Li, Q. Fu, A. X. Liu, Q. Li, and L. Mathy. Guarantee ip lookup performance with fib explosion. In *Proc. ACM SIGCOMM 2014*, volume 44, pages 39–50.

[47] M. K. Yoon, J. W. Son, and S. H. Shin. Bloom tree: A search tree based on bloom filters for multiple-set membership testing. In *INFOCOM, 2014 Proceedings IEEE*, pages 1429–1437, 2014.

[48] M. Yu, A. Fabrikant, and J. Rexford. Buffalo: bloom filter forwarding architecture for large organizations. In *ACM Conference on Emerging NETWORKING Experiments and Technology, CONEXT 2009, Rome, Italy, December*, pages 313–324, 2009.

[49] V. Zakhary, D. Agrawal, and A. E. Abbadi. Caching at the web scale. *Proceedings of the VLDB Endowment*, 10(12):2002–2005, 2017.

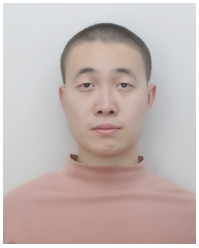
[50] L. Zdeborová and F. Krzakała. Phase transitions in the coloring of random graphs. *Physical Review E Statistical Nonlinear & Soft Matter Physics*, 76(3 Pt 1):031131, 2007.

[51] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.

[52] H. Zhong, Z. Li, Y. Xu, Z. Chen, and J. Cui. Two-stage index-based central keyword-ranked searches over encrypted cloud data. *Science China Information Sciences*, 63(3):1–3, 2020.

[53] H. Zhu, F. Xiao, L. Sun, R. Wang, and P. Yang. R-ttwd: Robust device-free through-the-wall detection of moving human with wifi. *IEEE Journal on Selected Areas in Communications*, 35(5):1090–1103, 2017.

AUTHORS



Zhaodong Kang
Kang Zhaodong has just graduated from Peking University, advised by Tong Yang. It is his first year to studying master in Peking University now. His research interests include network optimization, Bloom filters, and data storage structures.

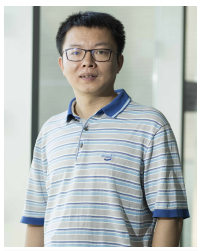
Jin Xu
Jin Xu is an undergraduate student of Peking University. His major is Data Science and Big Data Technology. His research interests include distributed algorithms and computational linguistics.

Wenqi Wang
Wenqi Wang is currently a senior undergraduate student of Peking University majoring in Computer Science. Her research interests include software engineering and software testing.

Jie Jiang
Jie Jiang is a postgraduate at Peking University, advised by Tong Yang. His research interests include indexing, data sketches, and data stream processing systems.

Shiqi Jiang
Shiqi Jiang is currently a sophomore student of Peking University. Her research interests include analysis and comparision of algorithms.

Tong Yang
Tong Yang received the PhD degree in computer science from Tsinghua University in 2013. He visited the Institute of Computing Technology, Chinese Academy of Sciences (CAS). Now he is an associate professor with Department of Computer Science and Technology, Peking University. His research interests include network measurements, sketches, IP lookups, Bloom filters, and KV stores. He published papers in SIGCOMM, SIGKDD, SIGMOD, NSDI, etc.



Bin Cui
Bin Cui is a professor in the School of EECS and Director of Institute of Network Computing and Information Systems, at Peking University. His research interests include database system architectures, query and index techniques, big data management and mining. He is a senior member of IEEE, member of ACM and distinguished member of CCF.

Tilman Wolf
Tilman Wolf is Senior Vice Provost for Academic Affairs, Interim Department Head of Biomedical Engineering, and Professor of Electrical and Computer Engineering at the University of Massachusetts Amherst. He is a co-author of the book "Architecture of Network Systems" and has published extensively in peer-reviewed journals and conferences. His research has been supported by grants from NSF, DARPA, and industry.