electronics

MDPI

Review

A Review of Cuckoo Filters for Privacy Protection and Their Applications

Yekang Zhao [1], Wangchen Dai [2], Shiren Wang [3], Liang Xi [3], Shenqing Wang [4] and Feng Zhang [4,*]

[1] Engineering Research Center of Digital Forensics, Ministry of Education, School of Computer Science, Nanjing University of Information Science and Technology, Nanjing 210044, China; zyk13812311766@163.com
[2] Research Center for Basic Theories of Intelligent Computing, Research Institute of Basic Theories, Zhejiang Lab, Hangzhou 310000, China; w.dai@my.cityu.edu.hk
[3] Beijing Institute of Computer Technology and Application, Beijing 100082, China; vivenrabbit@163.com (S.W.); xlcorn@163.com (L.X.)
[4] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 210016, China; shenqingtongxun@163.com
* Correspondence: njtongxun100@163.com; Tel.: +86-1732-611-7703

Abstract: As the global digitalization process continues, information is transformed into data and widely used, while the data are also at risk of serious privacy breaches. The Cuckoo filter is a data structure based on the Cuckoo hash. It encrypts data when it is used and can achieve privacy protection to a certain extent. The Cuckoo filter is an alternative to the Bloom filter, with advantages such as support for deleting elements and efficient space utilization. Cuckoo filters are widely used and developed in the fields of network engineering, storage systems, databases, file systems, distributed systems, etc., because they are often used to solve collection element query problems. In recent years, many variants of the Cuckoo filter have emerged based on ideas such as improving the structure and introducing new technologies in order to accommodate a variety of different scenarios, as well as a huge collection. With the development of the times, the improvement of the structure and operation logic of the Cuckoo filter itself has become an important direction for the research of aggregate element query.

Keywords: Cuckoo filter; aggregate element query; approximate membership query structure; hashing strategy; false positive rate; privacy protection

check for updates

Citation: Zhao, Y.; Dai, W.; Wang, S.; Xi, L.; Wang, S.; Zhang, F. A Review of Cuckoo Filters for Privacy Protection and Their Applications. *Electronics* **2023**, *12*, 2809. https://doi.org/10.3390/electronics12132809

Academic Editor: Djuradj Budimir

Received: 30 May 2023
Revised: 21 June 2023
Accepted: 21 June 2023
Published: 25 June 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

With the rapid development of the big data era, numerous new service models and applications have emerged [1]. These services and applications collect a large amount of information related to users while providing them with accurate and personalized services. Information is data that have meaning and value. It can contain text, images, sound, video, and other forms. When this information is transformed into digital data, it can be stored, processed, and transmitted [2,3]. The process of digitization involves the use of technical tools and methods to extract the characteristics of the information and encode it into a digital form that computers can understand and process [4]. Digitization transforms information into a form of data that makes them available for manipulation and utilization in computer systems. Through digitization, information can be analyzed, searched, shared, protected, and passed on to other systems or devices. The collected information often contains a large amount of private data, including phone numbers, ID numbers, and other personal identification information, as well as financial, medical, and health care information and other sensitive information [5,6]. This makes data among the most-valuable resources in the world [3,7]. The main risks associated with privacy breaches during data transformation include data leakage, data misuse, data association,

and data sharing [8–10]. The querying of data is the most-common operation when using data [11–13]. Because the data are very large, using the entire collection to query is very inefficient, and the process of traversal is also prone to data leakage [14]. A common processing method is to save the fingerprints of all elements in a collection in a hash table, using the one-way nature of the hash function to secure the data, while only the hash mapping is needed to determine whether the elements belong to the collection during the query. However, in maintaining the hash table representation both to store the fingerprints of all the elements and to save other structures that handle hash collisions, the space usage of the hash table is low. If the amount of data in the collection is very large, the hash table cannot be kept in memory. This results in queries that require iterative operations in memory and secondary storage, causing significant and unnecessary consumption. In some usage scenarios, certain errors are tolerated for aggregate element queries, a feature that motivates the creation of approximate member query (AQM) structures to answer such interactive query problems, of which the Cuckoo Filter (CF) is a typical representative.

The CF is derived from the Cuckoo hash algorithm, which maintains two hash tables, each using two different hash functions. When new data are inserted, the corresponding mapping location is calculated based on the fingerprint, and the fingerprint is present in one of these two locations. If both positions are occupied, a fingerprint is randomly kicked out and occupies its position, and the kicked-out data look for the corresponding position in the same way. By continuously kicking out data, eventually, all the data will find their corresponding locations. The CF, as an alternative to the Bloom Filter (BF) [11], preserves the advantages of the BF in handling problems such as near-membership queries while providing better query efficiency and space utilization on top of the BF. It can work with deletions, insertions, and queries without increasing the storage space. These good features have led to the use of CFs in several applications [15,16]. (Rao et al. proposed other more theoretical solutions for the BF, enabling it to have similar advantages [17]).

1.1. Advantages and Main Uses of Cuckoo Filters

CFs have four main advantages over Bloom filters, as follows:

(1) CFs support the dynamic addition and deletion of items.
(2) CFs have a higher lookup performance than standard Bloom filters, even when 95% of the space is occupied; CFs' lookup performance is still better than standard Bloom filters.
(3) Compared with some variants of Bloom filters (such as quotient filters), CFs are easier to achieve.
(4) With a target false positive rate of less than 3%, CFs use less space than Bloom filters in practical applications.

The main uses of CFs are as follows:

Determining whether an element exists in a collection: CFs can quickly determine whether an element exists in a collection by comparing the hash function with the fingerprint information. They can complete the query operation in a constant time, so they are widely used in scenarios that need to quickly determine whether an element exists, such as network security, database query, etc.

Improve query efficiency: CFs have the feature of fast query and can complete the query operation in constant time. Compared with traditional data structures such as hash tables or binary search trees, CFs have higher query efficiency, especially in the case of large-scale data collections. Therefore, they are widely used in scenarios that require efficient querying, such as search engines, network routing, etc.

Reduce conflict and optimize storage space: CFs adopt the Cuckoo hash algorithm, which can effectively reduce the occurrence of hash conflict and optimize the utilization of storage space. Through the reasonable design of the hash function and the way of storing fingerprint information, CFs can achieve a low conflict rate with a small storage space. Therefore, they are widely used in scenarios that require efficient use of storage space, such as the storage and indexing of large-scale data collections.

Support for distributed system applications: CFs can be extended to distributed system applications, and high availability and load balancing can be achieved by distributing data collections across multiple nodes. With reasonable data slicing and inter-node communication protocols, CFs can realize distributed query and update operations, thus supporting the application requirements of large-scale distributed systems.

Encryption and privacy protection: CFs use hash functions and fingerprint information to encrypt and obfuscate elements, thus protecting the privacy of data. They can transform sensitive information into irreversible fingerprints, ensuring that the original data cannot be restored and compromised. Therefore, CFs can play an important role in scenarios where data privacy needs to be protected.

1.2. Cuckoo Filters and Privacy Protection

The evolution of CFs is closely related to the overall development of digitization and data privacy protection. As digitization progresses, individuals and organizations are handling larger amounts of data containing more sensitive information [18,19]. At the same time, issues such as privacy leakage and data misuse are becoming increasingly prominent [20,21]. In order to effectively protect data privacy, related technologies and methods are constantly evolving.

The introduction of CFs can handle large-scale datasets more efficiently and alleviate the privacy leakage problem to some extent. By using Cuckoo filters, sensitive information can be desensitized and duplicate data detected and quickly filtered to reduce the risk of exposure of sensitive information.

In addition, the nature and features of CFs (e.g., spatially efficient and fast querying) make them an effective tool for large-scale data processing and privacy protection. They can be combined with other privacy-protection technologies, such as encryption, access control, and anonymization, to build a better data-privacy-protection system.

Thus, the evolution of CFs is closely related to the overall development of digitization and data privacy protection, and they provide a useful technical tool to meet the growing data challenges.

CFs can achieve a certain degree of privacy protection against the four privacy-breach-related risks in data transformation: data leakage, data misuse, data association, and data sharing. The following are a few ways to use CFs to mitigate privacy leakage issues:

(1) Data desensitization: Before converting or storing data, sensitive information can be desensitized using CFs. CFs can help detect the presence of known sensitive information, such as specific ID numbers, cell phone numbers, etc. If sensitive information is matched, the corresponding processing measures can be triggered to protect personal privacy.
(2) Preventing duplicate data: CFs can be used to detect duplicate data and avoid storing sensitive information repeatedly during data conversion. This helps reduce the chance and risk of sensitive information leakage.
(3) Fast filtering: CFs can be used to quickly filter out non-sensitive data, thereby reducing the amount of data that need to be further processed. With this filter, data that do not contain sensitive information can be quickly excluded, thus reducing the risk of exposure of sensitive information.

1.3. Recent Developments to Cuckoo Filters

Due to the above advantages, CFs can be adapted to most application scenarios and are the best tool to solve the "does the element belong to a collection" problem. CFs were first proposed in 2014 to solve the problem of the low efficiency of BF queries and low space utilization. CFs can be used for redis cache penetration, web applications, like BFs [22,23], push de-duplication for news clients, distributed databases [24–26], black and white lists, spell checking [27,28], and so on. CFs can also be used to improve the efficiency of other related applications, such as methods to protect the privacy of predictions in the federated learning phase [29], IoT smart cities [30–32], queries in wireless sensor networks [33,34],

and clustering algorithms [35,36]. Specific applications of CFs in several major scenarios can be found in Section 2.

However, CFs still have a number of problems:

(1) Deletion problem: Deletion only removes a copy of the fingerprint, and it is not certain that this copy of the fingerprint is the fingerprint of the element to be deleted. Furthermore, the deletion does not confirm whether the fingerprint exists in the CF. This situation can generate false positives.
(2) The insertion complexity is relatively high. As the number of inserted elements increases, the complexity becomes higher. When the bucket is full, the kick-out operation needs to be repeated, requiring the fingerprint of the proposed element to be recalculated.
(3) The size of the storage space must be an exponential multiple of 2, which creates a problem of low space utilization.
(4) The same element can be inserted at most kb-times (k refers to the number of hash functions, and b refers to the number of fingerprints that can be contained in the bucket, which can also be said to be the size of the bucket). Inserting the same item $kb + 1$-times will cause the insertion to fail.

The above problems will be more prominent when dealing with large amounts of data. After research and development, a large number of improvement schemes for the above problems have emerged. These improvement schemes are generally developed in four directions: the hashing strategy, the CF's own structure, the introduction of compression structure and filter integration; below are the recent advances of CFs in these directions.

The recent development in CFs in terms of the hashing strategy is Adaptive Cuckoo Filters (ACFs) [37]. Instead of using the standard CF's partial key Cuckoo hash, ACFs directly determine the bucket into which an element is inserted by the element's hash value (instead of only the fingerprint), thus allowing adaptive modification of the fingerprint when removing elements that cause false positives. This reduces the rate of false positives when dealing with large numbers of data queries.

In terms of improvements to the structure of the CF itself, the recent progress is the Marked Cuckoo Filter (MCF) [38], which adds an additional bit to each slot in the bucket to indicate the set to which the elements stored in that slot belong, for the multi-level ensemble scenario, which often occurs when processing large amounts of data.

In terms of introducing compression structures, the recent advancement in CFs is the XOR+ filter [39]. The core idea of the XOR+ filter is to group together the empty entries in a hash table and use a bit array to indicate the occupancy of the entries in the table and then transfer only those entries that are stored with elements during the transfer process, thus saving the memory space needed at runtime.

In terms of filter integration, the recent progress of CFs is Multiple Cuckoo Filter (MCF) [40]. In order to solve the problem that it is difficult to query whether a certain element exists in all data streams in a certain period of time in a huge amount of data, the MCF integrates multiple CFs so that each CF corresponds to a data stream to support concise membership queries for multiple data streams.

1.4. Motivation and Our Contributions

However, although there have been many studies on the improvement of the structure and operation logic of CFs, there is no systematic analysis and summary of these works in recent years in the literature. We believe that the improvement of CFs' structure and operation logic is an important research direction for the following reasons:

Improvement of query efficiency: The main goal of CFs is to quickly determine whether an element exists in the set or not. Therefore, optimizing the structure and operation logic of the Cuckoo filter can improve the query efficiency and reduce the query time and resource consumption. This is very important for the processing of large-scale datasets and efficient querying.

Improvement of conflict handling: CFs use hash functions to map elements to different locations, and conflict handling is required when there is a hash conflict. Improving the conflict handling algorithm of CFs can reduce the incidence of conflict and improve the accuracy and reliability of the data.

Optimization of storage space: The storage space of CFs is limited, so how to optimize the efficiency of storage space utilization is an important research direction. By improving the structure and operation logic of the Cuckoo filter, the storage space occupation can be reduced and the storage efficiency can be improved.

Distributed system support: With the widespread use of distributed systems, CF support in distributed environments has also become a research hotspot. Improving the structure and operation logic of the Cuckoo filter can allow adapting to the needs of distributed systems and improve the scalability and concurrency of the system.

Therefore, this paper provides an overview of the existing optimization schemes by classifying the different optimization methods from each scheme.

The Section 2 introduces Cuckoo hashing, the basic principles and operations of CFs, and the main application scenarios of CFs.

The Section 3 introduces the role of hash functions in CFs and the elaboration of CFs for optimizing hash functions.

The Section 4 presents a variety of improvement options for improving the CFs' structure.

The Section 5 describes the integration scheme for CFs and CFs with the introduction of compression structures.

The Section 6 compares the various options presented in the text and presents a vision for future development.

The Section 7 provides a summary of the article.

2. Standard Cuckoo Filter

2.1. Cuckoo Hash

Cuckoo hashing is an easy-to-implement hashing strategy. We assumed that all functions are independent and consistent. These assumptions are usually reasonable in practice [41]. The Cuckoo hash derives the mapped positions $h_1(x)$ and $h_2(x)$ about the elements using two mutually independent hash functions h_1 and h_2 [42]. It performs an insert element operation in the hash table. If one of the two mapped positions is free, the element is inserted directly into this position. (If both mapped positions are free, then insert the element in either position.) If both mapped positions have elements inserted, as in Figure 1, one of the elements is kicked out and the element to be inserted is inserted into this position. For the kicked-out element, it is inserted into the other of its two mapped positions. If there is still an element alreadykicked out and inserted into the next position, this will continue until a free position is inserted or the number of kicked-out elements reaches the threshold. The latter situation means that the table is full of elements and can no longer accept new ones. The insertion operation has failed, resulting in the need to expand the hash table's capacity.

In Figure 1, element E is computed by Cuckoo hashing to obtain the indexes $h_1(E)$ and $h_2(E)$ of two insertion positions. First, check the position corresponding to $h_1(E)$, and determine whether the position is already occupied by element A. Then, check the position corresponding to $h_2(E)$, and determine whether the position is occupied by element B. Therefore, kick out A or B randomly (in this case, kick out A), and insert E into the position. Element A is kicked out and looks for another mapping position of its own, and it turns out that the position happens to be occupied by B, so B is kicked out and A inserted in that position. the kicked-out element B then repeats the above process and finds another mapping position of its own, finds that the position is free, and inserts B in that position.

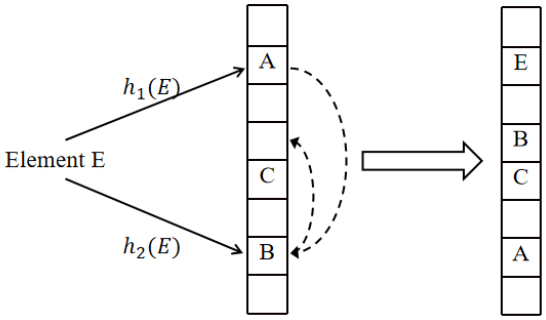

Figure 1. The element insertion operation of Cuckoo hash.

2.2. Standard Cuckoo Filter

The CF is a Cuckoo hash-based filter that outperforms most of the BF improvement schemes in terms of space usage, operational performance, and ease of implementation. Each CF location can store a fixed number of elements and stores the elements' fingerprints [43].

2.2.1. Basic Operation of the Cuckoo Filter

An important operation of the CF is insertion, a technique first introduced in a previous work [44]. However, since the computation of element fingerprints using a hash function is a one-way operation, it is not possible to compute two mapping positions of its corresponding elements for a particular fingerprint in the filter. The approach used by the CF for this is a partial key Cuckoo hash. For each fingerprint, only one hash function is used to compute a mapping position:

$$i_1 = hash(x). \tag{1}$$

The other mapping position is derived by the further calculation of the result of Equation (1):

$$i_2 = i_1 \oplus hash(f), \tag{2}$$

i_1 and i_2 are the indices of the two buckets computed. The equation of exchanging the positions of i_1 and i_2 in Equation (2) still holds according to the properties of the heterogeneous OR operation. That is, any one mapping position can be derived from a hash of another mapping position and an element fingerprint f by performing an XOR operation.

The specific fingerprint f is obtained by taking a certain number of bits by means of a hash function $f = fingerprint(x)$.

For CF query operations, take the process of querying a given element x as an example. The algorithm first takes a certain number of bits of x as the fingerprint f through the hash function $f = fingerprint(x)$ and then obtains the positions of the two buckets through Formulas (1) and (2); it compares f with the fingerprints in the bucket, and if any fingerprint in the bucket matches f, the CF returns true; if there are no fingerprints matching f in both buckets, the CF returns false. Through the above process, it can be seen that, in the absence of bucket overflow, there are no false positives.

The CF is like the counting Bloom filter, which removes inserted elements by removing the matching fingerprints, which are in the hash table; other filters that can perform similar deletion functions are more sophisticated than the CF [45,46]. For a given element x, the specific deletion process is to check two corresponding buckets; if the fingerprint in any bucket matches the fingerprint f of x, the copy of the matching fingerprint is deleted from that bucket.

2.2.2. Dimensions of the Drum

The maximum number of fingerprints that can fit in each bucket is called the bucket size, and changing the bucket size while keeping the total CF size constant leads to two consequences:

(1) Larger buckets improve table utilization (the larger the bucket, the greater the false positive rate is). The load factor α is 50% when using two hash functions when the bucket size is 1 (i.e., direct mapping of the hash table), but increases to 84%, 95%, and 98% when using bucket sizes of 2, 4, and 8, respectively [47].

(2) Larger buckets require longer fingerprints to maintain the same false alarm rate (i.e., the larger the bucket, the larger the fingerprint). When using larger buckets, more entries are checked for each lookup, thus increasing the probability of fingerprint conflicts.

2.3. Application Scenarios

The CF is mainly used in network engineering, caching systems, database systems, and distributed systems.

Network engineering: The CF can be used for fast matching of routing tables to speed up route lookup and forwarding operations. Reference [48] proposed a new filter called the length-aware Cuckoo filter (LACF) for faster IP lookups with limited additional storage. In addition, the CF has applications in wireless sensor networks. A wireless sensor network is a network consisting of a large number of wireless sensor nodes distributed in space for sensing information in the environment and transmitting it to a central node or other nodes [49–51]. In wireless sensor networks, an adversary may use critical data obtained from captured nodes to deploy a large number of cloned nodes in the network, thus affecting the network problem, i.e., cloned node attack [52,53]. A recent development in the solution to this problem was presented in [54], where a CF-based clone-node-detection algorithm for wireless sensor networks was proposed. Due to the simplicity and high efficiency of the CF in terms of the insertion and deletion of elements, the scheme in [54] had better detection time, power consumption, and detection accuracy, i.e., was 92% positive rate was achieved with 20% power consumption and a 98% detection rate.

Caching systems: In caching systems, the CF can be used to quickly determine whether an object is in the cache, thus accelerating cache hits and efficiency. Recent advances in CF research in this area are given in [55,56]. In [55], a CF-based hot-detection method with high spatiotemporal efficiency and support for deletion was proposed, as well as a cache replacement policy that combines the CF and an adaptive two-level LRU technique to obtain a significant improvement in the cache hit rate and a reduction in the time and space complexity. In [56], a CF-based scheme called PiPoMonitor was proposed to detect ping-pong patterns and prefetch specific cache lines to interfere with the adversary's cache probes to resist against cross-core cache attacks.

Database systems: Cuckoo filters can be used to speed up database query operations, for example, to determine whether an element exists in the database before querying it, thus avoiding unnecessary query operations. Reference [57] used the CF instead of traditional Bloom filters, thus improving the execution performance of query operations in big data warehouses. In addition, Reference [15] proposed an efficient CF-based scheme for database-driven cognitive radio networks (CRNs) that preserves the location privacy of secondary users (SUs), while allowing them to learn about the available channels in their vicinity; the latest advancement in this field was presented in [58], which proposed a new scheme using an object-level locking mechanism of the CF for improving the performance of very large object-storage-based database; the new scheme reduced the elapsed time to 60% while increasing the throughput to 171% compared to the scheme using a table-level locking mechanism.

Distributed systems: Cuckoo filters can be used for data consistency checking and de-duplication operations in distributed systems, e.g., in scenarios such as distributed caching, distributed storage, etc., to quickly determine whether data exist in other nodes in the cluster [59]. Reference [60] used the CF in the core lightweight client of the lightweight quantum-resistant distributed ledger protocol IOTA to avoid address reuse. A recent development in this area was presented in [61], which proposed a scheme applied to distributed big data systems, using the CF to improve the performance of lookups after data deletion; the CF was used to perform lookups before querying remote nodes, thus avoiding unneces-

sary network round-trip queries, and the scheme can improve the execution performance of query operations up to twice as much.

In addition, the CF has some applications related to smart cities. The term Smart City (Smart City) refers to the application of advanced technologies such as information technology and Internet of Things (IoT) technology to build an intelligent and sustainable city with the goal of improving the efficiency of city operations, optimizing the use of resources, and improving the quality of life of residents [21,62,63]. Among them, smart transportation is an important part of smart cities [64]. The scheme proposed in [65] used the CF to design big data generated in Vehicular Self-Organizing Networks (VANETs) for secure communication between vehicles and edge nodes. Reference [66] introduced Cuckoo filters in 5G vehicular networks to revoke malicious users to prevent re-attacks.

2.4. Some Related Studies on the Improvement of Cuckoo Filters

In 2016, Reference [48] proposed the Length-Aware Cuckoo Filter (LACF), which considers the problem of the popularity of collection elements and uses different insertion methods for elements with high and low popularity, effectively reducing the false alarm rate of the CF in scenarios such as IP address lookup. In 2017, Reference [67] proposed the d-Ary Cuckoo Filter (d-Ary CF), which solves the problem of the low space utilization of CFs in the face of a collection of useful and large numbers of elements by adjusting the bucket allocation strategy and sacrificing the efficiency of a small number of insertion queries. In 2017, Reference [68] proposed the Dynamic Cuckoo Filter (DCF), which extends the functionality of the CF, is the first data structure to support reliable element deletion and flexible structure expansion/compression, and enables the CF to be applied to dynamic collections. In 2018, Reference [69] proposed Position-Aware Cuckoo Filters (PACFs), which halved the false alarm rate of standard CF by telling in advance whether a fingerprint has been inserted into the first or second bucket. In 2019, Reference [70] proposed the Cuckoo Filter With an Integrated Bloom Filter (CFBF), which integrated a BF to enable insertion when no empty cell was found in the CF to be performed on the BF, thus reducing the insertion time. It supports removing all inserted elements. In a hardware implementation of the CF, the CFBF supports a large number of consecutive insertions. In 2019, Reference [71] proposed the Consistent Cuckoo Filter (CCF), which is able to be applied to many different scenarios by setting the parameters' flexibly. In 2020, Reference [72] proposed the Conditional Cuckoo Filter (Conditional CF), which added equational predicates to queries and introduced a new linking technique that enabled CFs to handle special sets determined by predicates and the insertion of duplicate keys. In 2022, Reference [40] proposed the Multiple Cuckoo Filter (Multiple CF), which implemented membership queries for multiple data streams by integrating multiple CFs. Various CF alternatives have also emerged during this period, resulting in enhancements in different aspects such as insertion, querying, space utilization, and false positive rate.

3. Cuckoo Filter for Improving Cuckoo Strategy

The hash function is an important part of CFs. It enables collection elements to be stored in filters in a very space-efficient manner and also unifies the data types and lengths of collection elements. Both short and long strings and large and small values can be turned into data of the same type and size by hash function calculation, thus making it easy to query and manage. In CFs, the output value of the hash function usually has two uses:

(1) As an address: Store an element in a bit vector table, and use the hash function to generate a number of stored random addresses.

(2) As the fingerprint of an element: When the data types between the elements of a CF-stored collection are inconsistent, the fingerprint of the element is usually obtained by first performing a hash operation on the element, and then, the fingerprint of the element is stored in the filter using the first method mentioned above. This ensures the diversity of data types stored in the CF and also achieves storage consistency while ensuring data privacy security to a certain extent.

However, no hash function is completely random, and as long as the mapping range is finite, there is bound to be the possibility of collision. The Cuckoo hash used in the standard CF uses eviction to solve the conflict problem, and as the number of inserted elements increases, the cost of insertion and lookup increases greatly, as well as the rate of false positives. This section describes two improvements to the CF hashing strategy: The LACF uses a different number of hash functions to store and search entries based on the prefix length and popularity of the routing entries. Adaptive Cuckoo Filters (ACFs) allow fingerprints to use different hash functions.

3.1. Length-Aware Cuckoo Filter

The main role of the LACF is to perform faster IP lookups. The LACF classifies elements into popular-length elements and unpopular-length elements based on the prevalence of their prefix length and uses this to distinguish their corresponding routing entries. For unpopular length elements, the LACF performs double insertion for them, so that, when querying a previous unpopular-length element, it needs to check two locations in the filter to determine its existence, thus reducing the false positive rate. For popular-length elements, the LACF inserts them only once, so the effect on the false positive rate can be ignored. Figure 2 shows a double insertion, where, for each element, the second insertion uses a separate set of hash functions (h_{1b}, h_{2b}). In Figure 2, the left half represents a normal primary insertion for a popular-length element, while the right half shows a secondary insertion for a non-popular-length element. In the right half of Figure 2, to insert a non-popular length element x into the filter, first, a set of partial key Cuckoo hash functions (h_1, h_2) is used to compute the two positions (a_1, a_2) for the first insertion performed on x and insert the fingerprint fp of element x into position a_1. Then, another independent set of Cuckoo hash functions (h_{1b}, h_{2b}) is used to compute the second set of insertion positions for x, i.e.,(a_{1b}, a_{2b}), and insert fp into position a_{2b}. The LACF actively adapts, identifies the element causing the false positives, and removes it, but still finds the element during the search, then inserts it again in a different way [48].

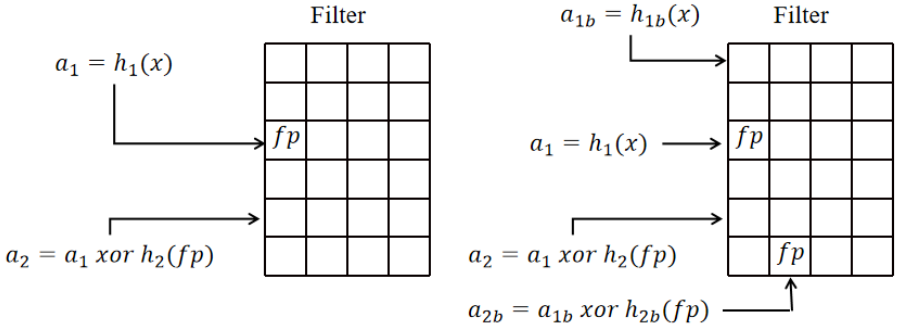

Figure 2. Element insertion in (**left**) Cuckoo filter and (**right**) double insertion.

The false positive rate of the CF is approximately $o(\frac{8}{2^f})$, where f represents the number of bits in the fingerprint, 8 is the number slots provided for each element in the filter, and o represents the occupancy rate, i.e., the percentage of filled positions in the filter to the total number of positions. For unpopular-length elements, the false positive rate of double insertion can be approximated as $o^2(\frac{8}{2^f})^2$ since both sets of hash functions have the possibility of false positives and their probabilities of false positives are independent of each other.

3.2. Adaptive Cuckoo Filters

For the set S of elements to be queried, the ACF stores the elements in it in a Cuckoo hash table. A copy of the Cuckoo hash table is also constructed to store the fingerprints corresponding to the elements in the set S, and this copy is used for the CF. The main feature of the ACF is that no partial key Cuckoo hash is used, and the ACF uses the hash of the complete elements instead of the fingerprints to determine the bucket to which each

element belongs [37]. When a query is performed on an element in S, a false positive occurs if a fingerprint in a bucket found during the query is the same as the fingerprint corresponding to that element. Therefore, when a positive result appears, the ACF checks the hash table, and if the fingerprint in the bucket does not correspond to the same element as the one being queried, it is determined to be a false positive. To remove false positives, the fingerprint associated with the element needs to be changed using a different fingerprint function. The ACF uses a method that identifies the element causing the false positive, removes it, but still finds the element when searching, and then inserts it again in a different way. This method ensures that no further false alarms occur when the same element is queried later, thus achieving the goal of reducing the false alarm rate.

4. Improved Structure of Cuckoo Filter

This section presents eight filter optimization schemes with an improved structure that essentially improve the structure of the CF, including the number of buckets in the filter and how bucket indexes are computed, the size of tables in the filter, eligibility testing of elements, finding connections between elements before use, and eviction strategies.

4.1. d-Ary Cuckoo Filter

To improve space utilization, one idea for CF improvement is to increase the number of candidate buckets corresponding to each element, and d-ary Cuckoo hashing takes advantage of this idea. Therefore, the CF is generalized to d-Ary CF to further improve the space utilization. However, it is difficult to add candidate buckets because only fingerprints are available for computing candidate positions. Therefore, a digitwise heteroskedastic operation based on d is introduced as a basis for computing d candidate buckets for each element in a round-robin fashion.

With 3-Ary CFs (three candidate buckets for each project), for example, you need to ensure that $A = AopBopBopB$ [48].

The base-3 digitwise XOR operation computes three candidate buckets for each element in a round-robin fashion. The operation op is XOR3, and XOR3 is equal to the digit mode operation in the base-3 digit system. The XOR3 computation rules are shown in Table 1.

Table 1. The rule of calculation of XOR_3.

	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

4.2. Consistent Cuckoo Filter

With several CF schemes previously described, it can be found that there is a dependency between the index of the bucket in which the elements are stored and the length of the filter, and the Index-Independent Cuckoo Filter (I2CF) eliminates this relationship. The core of the I2CF is to maintain a consistent hash ring, and when assigning buckets to elements, the I2CF can assign k candidate buckets for each element; the value of k is not fixed, so that the size can be adjusted by itself as needed. In this way, the I2CF achieves bucket-level capacity adjustment, allowing it to be used for dynamic collection presentation.

By organizing multiple I2CFs, a Consistent Cuckoo Filter (CCF) can be obtained. The capacity of the consistent Cuckoo filter is flexible, and when the set base increases or decreases suddenly, the CCF can be adjusted in time by merging underutilized I2CFs and adding unused I2CFs to the filter's capacity.

Within the CCF framework, the membership query can check all I2CF vectors. For I2CFi, the false positive rate is $\xi_i = 1 - (1 - \frac{1}{2^f})^{k_i \bullet b_i}$. This yields the global false positive rate of the CCF as $\xi_{CCF} = 1 - \prod_{i=0}^{s-1} (1 - \xi_i)$. To ensure the run-time false positive rate, the CCF

sets a threshold value for s. When s is about to reach the threshold value, in order to make room for new elements, the CCF will no longer add unused I2CFs, but choose to perform bucket-level capacity adjustment operations or perform compression operations by merging underused I2CFs. In this case, the value of s will not be added again, thus achieving control of the false positive rate. Consistent CF implements the design principle of dynamic set representation, but with a slightly higher complexity.

4.3. AniFilter

Advances in Non-Volatile Memory (NVM) technology have given persistent memory the advantage of high performance. Since AMQ is often used for persistent memory, CF, as a kind of AMQ, can also take advantage of the high performance of NVM to achieve persistence—AniFilter is such a special CF. Compared with plain CF, AniFilter achieves improved insertion throughput on NVM through spillable buckets and lookahead eviction, while achieving improved query throughput through a bucket primacy strategy.

In the AF, when there are not enough slots in a bucket, this bucket can borrow slots from its neighboring buckets. For example, when inserting a fingerprint FP into the i-th bucket ($b[i]$) in the AF, if all slots in $b[i]$ are full, then the AF does not kick a fingerprint from $b[i]$, but asks the buckets behind it if there is an empty slot; if there is a free slot in bucket $b[i+k]$, then insert the FP into that slot. At this point, we refer to the FP, bucket $b[i+k]$, and the slot where the FP is stored as the overflow fingerprint, overflow bucket, and overflow slot, respectively.

Under high load, fingerprints inserted in the CF face massive recycling, resulting in inefficient insertion. To solve this problem, the AF introduces lookahead eviction. Store the occupancy information for each bucket in an array of bits called occupancy flags. If the corresponding bucket is full, the entries in the array are set. The occupancy flag is referenced when an insert operation results in an eviction from the storage bucket. When selecting the FP to be evicted, we first list all the FPs in the bucket, then select one FP with a free spare bucket to evict based on the occupancy of those FPs' spare buckets. If none of the FPs have a free spare bucket, then one FP is randomly selected and evicted.

According to the Cuckoo hash, we know that, in the AF, there are two buckets available for each fingerprint of element x at the time of insertion. We refer to the bucket computed by h1(x) as the primary bucket and the other bucket (i.e., the one computed by h2(x)) as the secondary bucket. In both insertion and query, the algorithm looks for the primary bucket first and only takes the secondary bucket if the primary bucket is full. Therefore, in order to improve the throughput at query time, it is necessary to reduce the number of accesses to the auxiliary bucket and random accesses. To achieve this, the AF chooses to encode the eviction history of a bucket in terms of the order of the FPs stored in the bucket. Using a bucket with four slots as an example, one can determine whether a bucket has performed an eviction by comparing the FPs stored in the last two slots. If the FP stored in the fourth slot is larger than the FP stored in the third slot, then the bucket has not performed an eviction and the auxiliary bucket does not need to be queried.

Under continuous operation, the CF outperforms a host of other filters in terms of efficiency. However, under high load, the CF suffers from low insertion throughput. The AF, which is optimized on the basis of the CF, not only inherits the advantages of the CF, but also outperforms the CF and most other filters under high load [73].

4.4. Position-Aware Cuckoo Filters

In the CF, for each FP, there are two candidate buckets. It is easy to think that the false positive rate can be reduced if the bucket also stores information about the location of the FP in the bucket—i.e., which bucket this is for this FP. Positon-Aware Cuckoo Filters (PACFs) take advantage of this idea.

To achieve the above goal, it is easy to think of adding 1 bit to each FP to indicate the bucket it belongs to. For example, if the last 1 bit of the FP is 0, this means that the FP is located in the first bucket, i.e., bucket a_1; if the last 1 bit of the FP is 1, it means that the FP

is stored in bucket a_2. This seems to be able to put the false alarm rate to half of the original one. However, because the FP is increased by one bit, this again corresponds to increasing the false positive rate by a factor of two.

Since increasing the number of bits of the FP is not feasible, we might as well think differently; we can add a certain number of bits to the bucket to represent the location information of the FP stored in the bucket. As shown in Figure 3, for a bucket with 4 slots, the PACF adds two bits as the location information s and encodes the 4 slots as 0, 1, 2, and 3 from left to right. $s = 2$ in Figure 3, then for the FPs stored in Slots s to 4 (i.e., $fp(w)$ and $fp(z)$), this bucket is a_1; for the FPs stored in Slots 0 to $s - 1$ (i.e., $fp(x)$ and $fp(y)$), this bucket is a_2. Then, at query time, if this bucket is Bucket a1 for the queried element, then only the FP of the queried element needs to be compared with $fp(w)$ and $fp(z)$. This also reduces the false alarm rate to half of the original rate from a global view [69].

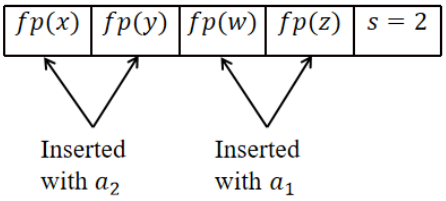

Figure 3. Marking of elements on a bucket in the position-aware Cuckoo filter.

4.5. Additive and Subtractive Cuckoo Filters

Huang et al. proposed a novel variant of the CF, the Additive and Subtractive Cuckoo Filter (ASCF), to improve the space utilization of the CF. The biggest improvement of the ASCF over the CF is the choice of replacing XOR operations with ADD/SUB operations to compute the bucket indexes. By getting rid of the XOR operation, the ASCF does not require the number of buckets to be a power of two, thus achieving higher space utilization while maintaining the high lookup and update performance of the CF.

In the ASCF, the m buckets in the hash table are equally divided into two blocks, and for an item x, its corresponding two candidate buckets are located within these two blocks; the following is the insertion process. If item x is to be inserted, then the ASCF first computes the fingerprint f_x of x within each of the two blocks, the index $h_0(x)$ of the bucket within Block 0, and the index $h_1(1)$ of the bucket within Block 1.

$$h_0(x) : f_x = G(x). \tag{3}$$

The $h_0(x)$ in (3) is the index of the bucket of item x in Block 0 and the index of the first candidate bucket of x in the range $[0, \ldots, m/2-1]$. f_x is the fingerprint of item x, ':' is the concatenator, and $G(x)$ is a hash corresponding to item x whose left part is the bucket index $h_0(x)$ and whose right part is the fingerprint f_x of x. Then, in Block 1, we use the addition operation to compute the bucket index of the second candidate bucket.

$$h_1(x) = (h_0(x) + H(f_x) mod\ m/2 + m/2). \tag{4}$$

Equation (4) shows the calculation of the index $h_1(x)$ for the buckets in Block 1. $H(f_x)$ ranges from $[0, \ldots, m/2-1]$, which is the number of buckets in each block. $h_1(x)$ ranges from $[m/2, \ldots, m-1]$.

After the indexes of the two candidate buckets are computed, if either of the two buckets is free, f_x is inserted into that bucket. If both buckets are full, an eviction is performed. The eviction process is to randomly insert f_x into bucket $h_0(x)$ or bucket $h_1(x)$ and then look for another candidate bucket of the evicted item. If the other candidate bucket is also full, the eviction process is repeated until a bucket with a free slot is found or the maximum number of evictions is reached. Equation (5) demonstrates the method to calculate the candidate buckets for the evicted items.

$$\begin{cases} j = (i + H(g)) mod\ m/2 + m/2 & if\ i \in block\ 0, \\ j = (i - H(g)) mod\ m/2 & if\ i \in block\ 1. \end{cases} \quad (5)$$

The g in Equation (5) is the fingerprint of the evicted item, and j is the index of the alternate bucket of the evicted item; we can see that the index of the alternate bucket is also obtained by the addition/subtraction operation. Compared to the CF, the ASCF not only maintains similar insertion and query throughput, but also reduces the space overhead by a factor of 1.9 [74].

4.6. Vacuum Filters

Vacuum Filters (VFs) are a set of AMQ data structures for items that support near-membership queries. Like most other AMQ data structures [5,11,75], vacuum filters may report false positive results.

VFs have a smaller space overhead and higher insertion and query throughput than other AMQ data structures. Thus, they are a more efficient and faster alternative to BFs and CFs. VFs also store fingerprints in hash tables, just like CFs. However, VFs have improved insertion and fingerprint eviction policies, thus enabling high load and data locality without limiting table size [76].

4.7. Conditional Cuckoo Filters

Conditional CFs allow setting membership tests for a given predicate on a pre-computed sketch. They are able to add equation predicates to queries. The filter also introduces a new linking technique that allows CFs to handle the insertion of duplicate keys [68]. This results in at least two significant benefits in joining processing. First, it allows predicate-specific filters to be applied to both the build side and the probe side of the connection. This increases the number of cases where the data structure created on the build side fits into main memory. Second, it allows predicates to be pushed down from one table to all other tables in the pass-through closure of the join graph. The number of tuples that the connection must handle is significantly reduced. Instead of storing keys and key pairs, conditional CFs store fingerprints or sketches of both and only the key fingerprints. The use of attribute sketches greatly improves the functionality of filters at a modest space cost.

Conditional CFs support two useful operations. Given an item x and a predicate P, they test x whether the item belongs to S_p. In other words, if there are matching rows in the input data, then it is S_p. As long as a predicate is given P, some variant of conditional CFs will return the set S_p of Cuckoo filters. Like other approximate set membership sketches, it maintains the property that false negative values cannot be returned.

Conditional CFs differ from Cuckoo hashes and filters in that the key may not be unique in conditional CFs and require techniques for handling copies. The tuples can share the same key, but have different properties. The use of linking techniques extends the Cuckoo hash table to make it robust to duplicate keys and allows high load factors to be achieved.

4.8. Marked Cuckoo Filter

One possible application area of CFs is for set representation in multi-set coordination problems. However, if the CF is used for multi-set representation, it is necessary to establish a correspondence between the elements stored in the buckets and the sets to which they belong [77]. To solve this problem, the Marked CF (MCF) adds a few bits in each slot for representing the affiliation information of the elements and calls these bits marker bits.

Similar to CFs, there are m buckets in MCFs and b slots in each bucket. However, unlike CFs, each slot has marker bits of length n in addition to the fingerprint bits of length f to indicate the adjunct information of the stored element, i.e., which set the element belongs to. The length n of the marker bits is also the number of sets that the MCF needs to

represent. If an element belongs to the set Si, then the *i*-th position of the marker bit of that element is 1, thus establishing the correspondence between the element and the set.

The insertion process of the MCF is similar to the CF. For the element x to be inserted, the MCF uses $h_1(x) = hash(x)\%m$ and $h_2(x) = h_1(x) \bigoplus (hash(\eta_x)\%m)$ computes the index of its corresponding two candidate buckets, where η_x is the fingerprint of element x [38]. If either of the two buckets has a free slot, η_x is inserted; if both buckets are full, a fingerprint is randomly evicted from either bucket to its spare bucket, and if the spare bucket is also full, the eviction is performed again until a bucket with a free slot is found or the number of evictions reaches a threshold.

5. Other Improved Structures

This section introduces two CF optimization schemes in terms of introducing the compression structure and filter integration, respectively. Introducing a compression model does not necessarily lead to a real space reduction, but it can effectively alleviate the bucket overflow problem and reduce the false positive rate. Using filter integration, you can effectively obtain the benefits of both the BF and CF while also having multiple filters to easily handle data from multiple streams.

5.1. Compression Structure

5.1.1. Morton Filters

The MF is a typical example of a CF using a compressed structure. The MF, like the CF, maintains a set of buckets with slots in each bucket for storing fingerprints, using two hash functions $H_1()$ and $H_2()$ to select candidate buckets for elements.

The biggest difference between the MF and CF is the use of a compressed storage format, the block. The size of the block is not fixed and is generally determined by the storage medium (cache, SSD, etc.) on which the MF is located. The MCF stores its own data in blocks, specifically a certain number of buckets and fingerprints in the buckets are stored in each block, as well as metadata; the metadata are used to recover the logical interpretation of the MF. Blocks are stored in a special storage structure, the block store. In the MF, blocks have three main components—Fingerprint Storage Array (FSA), Fullness Counter Array (FCA), and Overflow Tracking Array (OTA).

FSA: The FSA is an array that stores the fingerprints in the block. The slots in the block that have fingerprints stored in them are stored in the FSA in close succession in bucket order, while the free slots are all stored at the end of the buffer. Since only slots containing fingerprints are stored in the FSA, the number of slots in the FSA will be much less than the total number of slots in the block it is logically supposed to contain. This results in a lightly loaded filter while keeping the FSA full and saving storage space.

FCA: The FCA uses a fullness counter to count the number of fingerprints stored in each bucket in the block and uses this to encode all buckets in the block. With the help of the FCA, the MCF can implement in situ reading and writing to the buckets stored with sequence numbers in the FSA. For example, if we want to read the fingerprint information stored in bucket $FSA[3]$, then we only need to calculate the sum of the fullness values of the buckets $FSA[0]$, $FSA[1]$, and $FSA[2]$ and use them as offsets to quickly locate the position of the fingerprint in bucket $FSA[3]$. In addition, with the help of the FCA, the FSA does not have to store any free slots, thus saving the time for comparison with empty slots and improving the throughput of the filter [78].

OTA: The OTA can record the overflow of a block as a bit vector. When a fingerprint overflow occurs in a block, the OTA keeps track of the overflow by setting a bit. When querying a fingerprint in a block, the OTA can be used to determine whether the bucket to which the fingerprint belongs has overflowed or not, and thus decide whether it is necessary to query an alternate bucket. The OTA can help reduce the false alarm rate and increase the throughput.

The MF has higher throughput and lower space cost compared to the CF. In addition, the MF achieves a lower false alarm rate due to the reduced number of fingerprint comparisons.

5.1.2. XOR+ Filters

The XOR filter was proposed by Dietzfelbinger and Pagh and was originally a variant of the Bloomier filter. Compared to the Bloomier and Cuckoo filters, XOR filters have higher lookup efficiency while requiring lower memory overhead. Graf and Lemire proposed the XOR+ filter to further reduce the space overhead of XOR filters. XOR+ filters are more compact than XOR filters while maintaining a higher speedup than Bloomier filters.

In the XOR filter, the slots used to store the fingerprints are logically organized in the form of an array. The size of the array B, which is organized by all the slots, is determined by the size of the set S of elements. For security purposes, the size of B is generally slightly larger than the size of S. Specifically, if the capacity of the array B is represented by c, then $c \approx 1.23 \times |S|$. That is, there is about 19% of space in B that is empty. During the transfer, the XOR+ filter will not transfer these empty slots in order to improve performance. This is performed by first encoding a bit array based on the slot occupancy in Array B before transmission, with a "1" meaning the slot is occupied and a "0" meaning the slot is empty, and then sending the bit array. In this way, only those slots with fingerprints are transmitted, thus increasing the query speed.

The compression operation can also be performed at runtime of the XOR+ filter, thus saving runtime space. The XOR+ filter changes the original construction algorithm so that each of the three hash functions has a corresponding queue, and each hash function can map the elements into one third of the space of Table 1. The entries in the first two queues are then processed first until they are empty before the third queue is processed. This allows you to save space while maintaining good performance by moving most of the empty entries to the last third of Table 1 and then constructing ranked data structures for only that part.

5.2. Filter Integration

5.2.1. Cuckoo Filters with an Integrated Bloom Filter

Compared with the BF, the CF has a lower false alarm rate, and the CF supports deletion operations, so the CF can be a good replacement for the BF in most cases. However, the CF also has problems. The CF has more complex insertion operations than the BF. The CF has poor performance when inserting at high occupancy. It is difficult for the CF to support high speed and a large number of consecutive insertions. For this reason, Reviriego et al. proposed the CFBF, which is a scheme to integrate the BF into the CF.

In the CFBF, in addition to four slots in each bucket, there is an additional Bloom filter bit bf, as shown in Figure 4. In the following, we describe the insertion process in the CFBF. For a new element x, x can be inserted either into the CF or into the BF, and the exact choice is determined by the algorithm. The purpose of integrating the BF in the CF is to reduce the time consumed by performing a large number of consecutive insertions in the CF at high occupancy, so the CFBF sets a threshold for the number of insertions t. First, an insertion is performed into the standard CF, and if the number of iterations of the insertion operation reaches t, then the next BF insertion is performed [70]. The process of inserting x into the BF is to first calculate $h_1(x)$ and $h_2(x)$ to obtain two buckets a_1 and a_2 and then set the bf bits of both buckets a_1 and a_2 to 1 (in order to avoid conflicts), thus completing the insertion.

bf	$cell-1$	$cell-2$	$cell-3$	$cell-4$

Figure 4. Structure of a bucket in the proposed CFBF.

From the above insertion process, we can deduce the lookup process of the CFBF. For the element q with a query, the fingerprint FP_q of q and the two buckets a_1 and a_2 are calculated first. If the fingerprint fp matching FP_q is found in a_1 or a_2, the lookup succeeds and returns SUCCESS, which is the case of the element in the CF; if the bf bits of both a_1 and a_2 are 1, SUCCESS is returned, which is the case of the element in BF. In addition, to remove an element from the BF, simply set the bf bits of both buckets to 0.

The evaluation showed that the CFBF can reduce the insertion time in the worst case to one-tenth of the original one and improve the average insertion efficiency in the high-occupancy case by more than 10-times the original one, compared with the standard CF. The CFBF supports the operation of deleting the inserted elements.

5.2.2. Multiple Cuckoo Filter

In the era of big data, especially after the popularity of smartphones, the sources of information that can provide data streams are becoming more and more abundant. The information provided by a single data stream is often rather one-sided, and it is difficult to meet the needs of relevant personnel. The information of multiple data streams is often correlated, so the integration of multiple data streams can help us obtain more comprehensive information, which requires filters that can be used for multi-dimensional element membership queries.

To address the above problem, there are some studies that chose to improve on Bloom filters, the core idea of which is to create a Bloom filter for each dimension, thus enabling efficient membership queries. However, the existing schemes fail to address the problem that it is impossible to determine whether an element exists in multiple data streams within a certain time period. To solve this problem, Hu et al. proposed the Multiple Cuckoo Filter (MCF) [40] based on existing schemes.

The MCF assigns a standard Cuckoo filter to each data stream, thus enabling the decomposition of membership queries on a data stream-by-stream basis, with each Cuckoo filter responsible for querying a single data stream. For different types of data streams, the MCF also introduces a window mechanism, where the window size can be different for each data stream. The window is used to split each data stream according to the time period, and the fingerprints in the window are inserted into the corresponding CF of the data stream. Since the head element of the window needs to be deleted and a new element inserted at the end when the sliding window is dynamically changed, a queue is used as the data structure of the sliding window. The motion direction of the sliding window is shown in Figure 5.

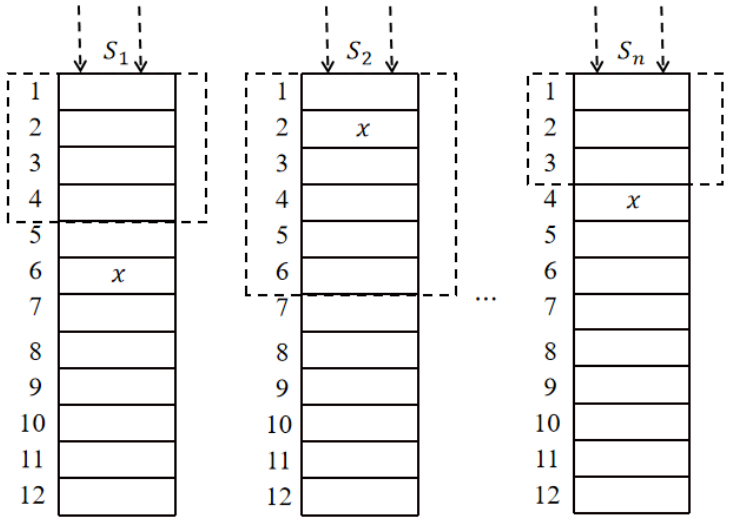

Figure 5. MCF.

As shown in Figure 5, S_1, S_2, and S_n are different data streams, and the dashed boxes indicate the sliding windows of each data stream with the sliding direction from top to bottom. In Figure 5, S_1 has a window size of 4, S_2 has a window size of 6, and S_n has a window size of 3. The elements in the window are inserted into the corresponding CF of each data stream according to the standard CF insertion process. In Figure 5, it is necessary to find element x in all data streams. For each data stream, the sliding window starts from the starting position of the data stream to be processed and divides the data stream

according to the window size and sliding interval. If the current Cuckoo filter is empty, the fingerprint of all records in the window is inserted into its own Cuckoo filter; if it is not empty, the Cuckoo filter is cleared, and then, the data in the window are inserted into the Cuckoo filter. Check whether x exists in the current window. If so, the current window stops sliding and returns true; otherwise, determine whether the current window has slid to the end of the current data stream, and return false if so. Finally, determine whether each data stream returns true. If all data streams return true, then x does exist in each data stream. However, if any of the streams return false, this means that element x does not exist in all streams for a certain period of time.

Since it is necessary to determine whether an element exists in more than one data stream in a certain time period, the query for each element needs to retrieve all CFs. Assuming that there are s data streams and filters in total and the false alarm rate of each CF is ϵCF, then the false alarm rate of the MCF does not exceed $1-(1-\epsilon CF)^s$. The false positive rate of the MCF considering the dynamic window is

$$\epsilon MCF = 1-(1-\epsilon CF)^{s\cdot\lfloor (m-w+1)/k\rfloor+1} \approx \frac{2bs(m-w+1)}{k\cdot 2^f}. \quad (6)$$

The $s\cdot\lfloor (m-w+1)/k+1\rfloor$ in (6) is the total number of comparisons performed; w is the size of the sliding window; m is the number of records contained in each data stream; k is the number of elements being moved each time. b represents the maximum number of fingerprints that can be contained in each bucket of each CF, and f represents the length of each fingerprint. It can be seen that, as with the CFs, the fingerprint length f has a relatively large impact on the false alarm rate of the MCF.

6. Analysis and Exploration of Improvement Schemes

6.1. Scheme Analysis

The CF has had dozens of optimizations and variations since it was proposed in 2014, and new schemes have been proposed in recent years. Table 2 provides a comparative analysis of the 15 existing typical CF optimization schemes mentioned in this paper and the standard CF in terms of four generality metrics: query, insertion performance, space utilization, false alarm rate, and five aspects of usage scenarios, where $\surd$ indicates that this performance is optimized, $\times$ indicates that this performance is sacrificed, and $-$ indicates that this performance is unchanged or not comparable. Of these solutions, most are mainly proposed for specific scenarios or as alternatives to the CF.

Table 2. Comparison of improvement schemes of the CF.

Filter Name	Performance Features				Application Scenarios
	Inquiry	Insertion	Space Use	False Positive Rate	
Standard CF	–	–	–	–	Proximate membership search
LACF	–	×	–	√	IP address Lookup
ACF	–	–	–	√	Proximate membership search
d-Ary CF	×	×	√	–	Very large collection
Consistent CF	√	–	√	–	Flexible parameter adjustment
AF	√	√	–	–	Proximate membership search
PACF	–	–	–	√	IP lookup and information retrieval
ASCF	–	–	√	–	Proximate membership search
VF	√	√	–	–	Proximate membership search
Conditional CFs	–	–	–	–	Join processing and other sets determined by predicates
Marked CF	–	–	–	–	Multi-party collection
Multiple CF	–	–	–	√	Membership queries for multiple data streams
CFBF	–	√	–	√	Achieve continuous insertion of CFs
DCF	√	√	√	–	Dynamic set
MF	√	√	√	√	Proximate membership search
XF	√	×	√	–	Proximate membership search

This paper first introduced two CF improvement schemes that optimize the hash policy, essentially improving the CF operation logic. The improvement for hash functions is a performance improvement for all application scenarios of CFs. Although CFs have outstanding performance in query and delete operations, element insertion will require an eviction operation due to hash collision, thus shifting the position of a large number of irrelevant elements and resulting in wasted performance. The two schemes proposed in this paper both use the hash function adaptively for different fingerprint entries, which can effectively avoid the occurrence of hash collisions, which essentially solves the problem of low CF insertion efficiency. Among them, the LACF also considers the use of different insertion methods according to the prefix length prevalence of the routing entries, and this scheme is suitable for data with an obvious division of the prefix length, such as IP addresses.

The main CF optimization solutions for structural improvement are to increase the functionality of CFs or to continue optimizing CFs to achieve full replacement. Among them, conditional CFs handle collections of data determined by a set of predicates and introduce new concatenation techniques that allow CFs to handle the insertion of duplicate keys. Marked CFs target collections where there is some association between collection elements. Marked CFs need to store the association between elements by jointly considering fingerprints and marker fields in slots. Marked CF naturally supports multi-party set representation.

The CF improvement scheme of the compression structure was introduced mainly to optimize its performance. The compression of filters using existing compression techniques has theoretically demonstrated that such structural compression not only reduces space usage, but also reduces the query misclassification rate of filters. Filter integration for CFs, the inheritance of a BF, and the integration of a set of CFs are all designed to obtain the advantages of the integrated part to compensate for the shortcomings of a single CF. Among them, the BF can take advantage of its higher insertion efficiency than the CF to improve the insertion efficiency and achieve continuous insertion of data. The query of multiple data streams is divided into logical individual data streams, and a CF is provided for each data stream to realize the processing of multiple data streams by the CF.

6.2. Future Development Prospects

After continuous improvement and optimization, the optimized solutions of CFs have been greatly improved in terms of space utilization, false alarm rate, etc. However, in the face of a wide range of usage scenarios, general-purpose CFs inevitably lack pertinence. Therefore, it will be a hot issue for future research to further optimize existing schemes to adapt them to a wide range of application scenarios. We see the potential of CFs for a wide range of applications in machine learning and artificial intelligence to improve data quality, improve model performance, and discover hidden patterns and features. Artificial intelligence models are usually more sensitive to high-quality data, and CFs can help remove noise, smooth data, and improve data quality [79]. This improves the training effectiveness and performance of AI models [80]. CFs are able to identify and extract important features and patterns in time series data. Combining them with AI can enhance the model's ability to learn these features and improve the accuracy and generalization of the model [81]. However, the various current CF schemes have not yet been able to implement these ideas well, so improvements in the structure and operational logic of CFs are needed to further explore the potential of CFs.

The optimization of the hash strategy is a critical component of CFs; whether or not the mapping is uniform is related to the efficiency of CFs' space utilization, and collision in finite space is a direct cause of CFs' query misclassification rate. Although there must be collisions of hash strategies in a finite space, its further exploration is an important direction for the future.

None of the current optimizations for CFs are considered from a privacy perspective. It is also a worthwhile direction to consider whether the hash function can be replaced or other more-secure and efficient encryption schemes can be incorporated.

With the increasing size of data and the gradual increase of interactions, it is difficult to avoid the scenario of multi-collection data processing. There are still relatively few relevant improvement solutions for multiple collection element queries.

In addition to improving the structure and operation logic of the CF itself, combining it with new technologies is also an important improvement direction for CFs. For example, we believe that CFs can be combined with neural networks so that CFs can gain new advantages.

A Neural Network (NeN) is a computational model consisting of a large number of artificial neurons (also called nodes or units), inspired by biological nervous systems [82,83]. It is widely used in the field of machine learning and artificial intelligence. Optimization of Cuckoo filters using neural networks can provide the following benefits:

(1) Neural networks can learn more complex patterns and features, thus improving the accuracy of CFs. By training the neural network, they can identify and filter more accurate data and reduce the number of misclassifications and omissions.
(2) The neural network can automatically adjust the weights and model structure according to the changes in input data to adapt to different data distributions and features. This allows the CF to adapt and process better with better generalization ability when facing new data.
(3) Neural networks are able to handle nonlinear relationships and complex features, which can capture more semantic information and contextual associations. For text data, neural networks can understand features at the lexical, syntactic, and semantic levels to better distinguish between normal content and malicious attacks.
(4) The optimized CF can take advantage of the neural network to perform filtering and judgment faster with the advantage of parallel computing. This is important for data stream processing in real-time scenarios to improve response time and processing efficiency.
(5) Neural networks are very scalable and can be extended to multi-layer, multi-type network structures to accommodate more complex data analysis needs. This allows CFs to handle a wider range of data types and tasks, with more powerful functions and application potential.

In summary, the optimization of CFs using neural networks can improve accuracy, be adaptable, handle complex features, be real-time, and have good scalability. This will enhance CFs in terms of data processing and security protection, providing a better experience and protection for users.

With the development of the big data era, CFs still have a wide scope of use due to their compact space usage and efficient operation. They can also be optimized in all parts and improved for specific application scenarios. Therefore, CFs will remain a major topic in the future in the research area of high-performance queries.

7. Conclusions

With the increasing scale of data, more and more private information is transformed into data. Many scenarios of collection element queries need to be carried out in massive data with increasing requirements for performance, as well as privacy. With the emergence of different CF improvement schemes, the insertion and lookup performance of CFs is continuously improved, and space utilization is gradually improved while also maintaining a low false alarm rate. This makes CFs the most-common tool for element membership queries. In this paper, we introduced numerous CF improvement schemes for different scenarios and CF alternatives, reviewed numerous CF improvement schemes from four perspectives: the hashing strategy, the CF structure itself, the introduction of the compression structure, and filter integration, and compared and analyzed the main

performance metrics of CFs to provide a reference for possible future research directions of CFs.

Author Contributions: Y.Z., W.D. and S.W. (Shiren Wang) were responsible for conceptual analysis, methodological analysis, and writing of the original draft. S.W. (Shenqing Wang) and L.X. were responsible for the thesis revision and review. F.Z. was responsible for the review, supervision, and project administration. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Key R&D Program of China (Grant No. 2021YFB3101104), and it was also supported by the National Natural Science Foundation of China (Grant No. 62072249). This work was also supported by the National Key R&D Program of Guangdong Province (Grant No. 2020B0101090002) and the Natural Science Foundation of Jiangsu Province (Grant No. BK20200418, BE2020106).

Data Availability Statement: No new data were created nor analyzed in this study. Data sharing is not applicable to this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, J.; Meng, X. Survey on privacy-preserving machine learning. *J. Comput. Res. Dev.* **2020**, *57*, 346–362.
2. Srivastava, J.; Maddheshiya, S. Retrieving the missing data from different incomplete soft sets. *3c Empres. Investig. Pensam. Crítico* **2022**, *11*, 104–114. [CrossRef]
3. Ren, Y.; Huang, D.; Wang, W.; Yu, X. BSMD: A blockchain-based secure storage mechanism for big spatio-temporal data. *Future Gener. Comput. Syst.* **2023**, *138*, 328–338. [CrossRef]
4. Paikrao, P.; Doye, D.; Bhalerao, M.; Vaidya, M. Verification of Role of Data Scanning Direction in Image Compression using Fuzzy Composition Operations. *3c Tecnol. Glosas Innov. Apl. Pyme* **2022**, *11*, 38–49. [CrossRef]
5. Yan, Y.; Ma, M.; Jiang, H. An efficient privacy preserving 4PC machine learning scheme based on secret sharing. *J. Comput. Res. Dev.* **2022**, *59*, 2338–2347.
6. Fang, L.; Li, Y.; Yun, X.; Wen, Z.; Ji, S.; Meng, W.; Cao, Z.; Tanveer, M. THP: A novel authentication scheme to prevent multiple attacks in SDN-based IoT network. *IEEE Internet Things J.* **2019**, *7*, 5745–5759. [CrossRef]
7. Yu, X.; Zhu, S.; Ren, Y. Continuous trajectory similarity search with result diversification. *Future Gener. Comput. Syst.* **2023**, *143*, 392–400. [CrossRef]
8. Ge, C.; Susilo, W.; Baek, J.; Liu, Z.; Xia, J.; Fang, L. A verifiable and fair attribute-based proxy re-encryption scheme for data sharing in clouds. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2907–2919. [CrossRef]
9. Wu, Q.; Xi, L.; Wang, S.; Ji, S.; Wang, S.; Ren, Y. Verifiable Delay Function and Its Blockchain-Related Application: A Survey. *Sensors* **2022**, *22*, 7524. [CrossRef]
10. Ren, Y.; Leng, Y.; Cheng, Y.; Wang, J. Secure data storage based on blockchain and coding in edge computing. *Math. Biosci. Eng.* **2019**, *16*, 1874–1892. [CrossRef]
11. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. [CrossRef]
12. Ge, C.; Susilo, W.; Liu, Z.; Xia, J.; Szalachowski, P.; Fang, L. Secure keyword search and data sharing mechanism for cloud computing. *IEEE Trans. Dependable Secur. Comput.* **2020**, *18*, 2787–2800. [CrossRef]
13. Ren, Y.; Leng, Y.; Qi, J.; Sharma, P.K.; Wang, J.; Almakhadmeh, Z.; Tolba, A. Multiple cloud storage mechanism based on blockchain in smart homes. *Future Gener. Comput. Syst.* **2021**, *115*, 304–313. [CrossRef]
14. Lu, J.; Zhu, L.; Gao, W. Remarks on bipolar cubic fuzzy graphs and its chemical applications. *Int. J. Math. Comput. Eng.* **2023**, *1*, 1–9.
15. Grissa, M.; Yavuz, A.A.; Hamdaoui, B. Cuckoo filter-based location-privacy preservation in database-driven cognitive radio networks. In Proceedings of the 2015 World Symposium on Computer Networks and Information Security (WSCNIS), Hammamet, Tunisia, 19–21 September 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 1–7.
16. Gupta, V.; Breitinger, F. How Cuckoo filter can improve existing approximate matching techniques. In Proceedings of the Digital Forensics and Cyber Crime: 7th International Conference, ICDF2C 2015, Seoul, Republic of Korea, 6–8 October 2015; Revised Selected Papers 7; pp. 39–52.
17. Pagh, A.; Pagh, R.; Rao, S.S. An optimal bloom filter replacement. In Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Vancouver, BC, Canada, 23–25 January 2005; pp. 823–829.
18. Zhang, J.; Peng, S.; Gao, Y.; Zhang, Z.; Hong, Q. APMSA: Adversarial Perturbation Against Model Stealing Attacks. *IEEE Trans. Inf. Forensics Secur.* **2023**, *18*, 1667–1679. [CrossRef]
19. Jiang, H.; Wang, M.; Zhao, P.; Xiao, Z.; Dustdar, S. A utility-aware general framework with quantifiable privacy preservation for destination prediction in LBSs. *IEEE/ACM Trans. Netw.* **2021**, *29*, 2228–2241. [CrossRef]

20. Li, B.; Zhou, X.; Ning, Z.; Guan, X.; Yiu, K.F.C. Dynamic event-triggered security control for networked control systems with cyber-attacks: A model predictive control approach. *Inf. Sci.* **2022**, *612*, 384–398. [CrossRef]
21. Ma, J.; Hu, J. Safe consensus control of cooperative-competitive multi-agent systems via differential privacy. *Kybernetika* **2022**, *58*, 426–439. [CrossRef]
22. Broder, A.; Mitzenmacher, M. Network applications of bloom filters: A survey. *Internet Math.* **2004**, *1*, 485–509. [CrossRef]
23. Byers, J.; Considine, J.; Mitzenmacher, M.; Rost, S. Informed content delivery across adaptive overlay networks. *ACM SIGCOMM Comput. Commun. Rev.* **2002**, *32*, 47–60. [CrossRef]
24. Li, Z.; Ross, K.A. Perf join: An alternative to two-way semijoin and bloomjoin. In Proceedings of the Fourth International Conference on Information and Knowledge Management, Baltimore, MD, USA, 28 November–2 December 1995; pp. 137–144.
25. Mullin, J.K. Optimal semijoins for distributed database systems. *IEEE Trans. Softw. Eng.* **1990**, *16*, 558–560. [CrossRef]
26. Mullin, J.K. Estimating the size of a relational join. *Inf. Syst.* **1993**, *18*, 189–196. [CrossRef]
27. Mullin, J.K.; Margoliash, D.J. A tale of three spelling checkers. *Softw. Pract. Exp.* **1990**, *20*, 625–630. [CrossRef]
28. McIlroy, M. Development of a spelling list. *IEEE Trans. Commun.* **1982**, *30*, 91–99. [CrossRef]
29. Fu, A.; Zhang, X.; Xiong, N.; Gao, Y.; Wang, H.; Zhang, J. VFL: A verifiable federated learning with privacy-preserving for big data in industrial IoT. *IEEE Trans. Ind. Inform.* **2020**, *18*, 3316–3326. [CrossRef]
30. Kumar, P.; Kumar, R.; Srivastava, G.; Gupta, G.P.; Tripathi, R.; Gadekallu, T.R.; Xiong, N.N. PPSF: A privacy-preserving and secure framework using blockchain-based machine-learning for IoT-driven smart cities. *IEEE Trans. Netw. Sci. Eng.* **2021**, *8*, 2326–2341. [CrossRef]
31. Chiu, W.Y.; Meng, W. EdgeTC-a PBFT blockchain-based ETC scheme for smart cities. *Peer-to-Peer Netw. Appl.* **2021**, *14*, 2874–2886. [CrossRef]
32. Meng, F.; Xiao, X.; Wang, J. Rating the crisis of online public opinion using a multi-level index system. *arXiv* **2022**, arXiv:2207.14740.
33. Yao, Y.; Xiong, N.; Park, J.H.; Ma, L.; Liu, J. Privacy-preserving max/min query in two-tiered wireless sensor networks. *Comput. Math. Appl.* **2013**, *65*, 1318–1325. [CrossRef]
34. Wang, J.; Ju, C.; Gao, Y.; Sangaiah, A.K.; Kim, G.J. A PSO based energy efficient coverage control algorithm for wireless sensor networks. *Comput. Mater. Contin* **2018**, *56*, 433–446.
35. Chen, Y.; Zhou, L.; Pei, S.; Yu, Z.; Chen, Y.; Liu, X.; Du, J.; Xiong, N. KNN-BLOCK DBSCAN: Fast clustering for large-scale data. *IEEE Trans. Syst. Man Cybern. Syst.* **2019**, *51*, 3939–3953. [CrossRef]
36. Ge, C.; Susilo, W.; Baek, J.; Liu, Z.; Xia, J.; Fang, L. Revocable attribute-based encryption with data integrity in clouds. *IEEE Trans. Dependable Secur. Comput.* **2021**, *19*, 2864–2872. [CrossRef]
37. Mitzenmacher, M.; Pontarelli, S.; Reviriego, P. Adaptive Cuckoo Filters. *ACM J. Exp. Algorithmics* **2020**, *25*, 1–20 [CrossRef]
38. Luo, L.; Guo, D.; Zhao, Y.; Rottenstreich, O.; Ma, R.T.; Luo, X. MCFsyn: A multi-party set reconciliation protocol with the marked Cuckoo filter. *IEEE Trans. Parallel Distrib. Syst.* **2021**, *32*, 2705–2718. [CrossRef]
39. Graf, T.M.; Lemire, D. XOR filters: Faster and smaller than bloom and Cuckoo filters. *J. Exp. Algorithmics (JEA)* **2020**, *25*, 1–16. [CrossRef]
40. Hu, Z.; Wu, M.; Fan, X.; Wang, Y.; Xu, C. MCF: Towards Window-Based Multiple Cuckoo Filter in Stream Computing. In Proceedings of the Big Data—BigData 2020: 9th International Conference, Held as Part of the Services Conference Federation, SCF 2020, Honolulu, HI, USA, 18–20 September 2020; pp. 101–115.
41. Mitzenmacher, M.; Vadhan, S.P. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proceedings of the SODA*; Citeseer: Philadelphia, PA, USA, 2008; Volume 8, pp. 746–755.
42. Pagh, R.; Rodler, F.F. Cuckoo hashing. *J. Algorithms* **2004**, *51*, 122–144. [CrossRef]
43. Eppstein, D. Cuckoo filter: Simplification and analysis. *arXiv* **2016**, arXiv:1604.06067.
44. Fan, B.; Andersen, D.G.; Kaminsky, M. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), Lombard, IL, USA, 2–5 April 2013; pp. 371–384.
45. Fan, L.; Cao, P.; Almeida, J.; Broder, A.Z. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.* **2000**, *8*, 281–293. [CrossRef]
46. Hua, W.; Gao, Y.; Lyu, M.; Xie, P. Research on Bloom filter: A survey. *J. Comput. Appl.* **2022**, *42*, 1729.
47. Fan, B.; Andersen, D.G.; Kaminsky, M.; Mitzenmacher, M.D. Cuckoo filter: Practically better than bloom. In Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, Sydney, Australia, 2–5 December 2014; pp. 75–88.
48. Kwon, M.; Reviriego, P.; Pontarelli, S. A length-aware Cuckoo filter for faster IP lookup. In Proceedings of the 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, USA, 10–14 April 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1071–1072.
49. Wang, J.; Gao, Y.; Liu, W.; Sangaiah, A.K.; Kim, H.J. An intelligent data gathering schema with data fusion supported for mobile sink in wireless sensor networks. *Int. J. Distrib. Sens. Netw.* **2019**, *15*, 1550147719839581. [CrossRef]
50. Ren, Y.; Zhu, F.; Sharma, P.K.; Wang, T.; Wang, J.; Alfarraj, O.; Tolba, A. Data query mechanism based on hash computing power of blockchain in internet of things. *Sensors* **2019**, *20*, 207. [CrossRef] [PubMed]

51. Shi, Y.; Li, H.; Fu, X.; Luan, R.; Wang, Y.; Wang, N.; Sun, Z.; Niu, Y.; Wang, C.; Zhang, C.; et al. Self-powered difunctional sensors based on sliding contact-electrification and tribovoltaic effects for pneumatic monitoring and controlling. *Nano Energy* **2023**, *110*, 108339. [CrossRef]
52. Wang, J.; Gu, X.; Liu, W.; Sangaiah, A.K.; Kim, H.J. An empower hamilton loop based data collection algorithm with mobile agent for WSNs. *Hum.-Centric Comput. Inf. Sci.* **2019**, *9*, 18. [CrossRef]
53. Sankar Chatterjee, P.; Roy, M. Lightweight cloned-node detection algorithm for efficiently handling SSDF attacks and facilitating secure spectrum allocation in CWSNs. *IET Wirel. Sens. Syst.* **2018**, *8*, 121–128. [CrossRef]
54. Sajitha, M.; Kavitha, D.; Reddy, P.C. An Optimized Clone Node Detection in WSN Using Cuckoo Filter. *SN Comput. Sci.* **2023**, *4*, 167. [CrossRef]
55. Wang, Y.; Yang, Y.; Qiu, X.; Ke, Y.; Wang, Q. CCF-LRU: Hybrid storage cache replacement strategy based on counting Cuckoo filter hot-probe method. *Appl. Intell.* **2022**, *52*, 5144–5158. [CrossRef]
56. Yuan, F.; Wang, K.; Hou, R.; Li, X.; Li, P.; Zhao, L.; Ying, J.; Awad, A.; Meng, D. PiPoMonitor: Mitigating cross-core cache attacks using the auto-Cuckoo filter. In Proceedings of the 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 1–5 February 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1697–1702.
57. Mosharraf, S.I.M.; Adnan, M.A. Improving Query Execution Performance in Big Data using Cuckoo Filter. In Proceedings of the 2018 IEEE International Conference on Big Data (Big Data), Seattle, WA, USA, 10–13 December 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 1079–1084.
58. Krishna, R.S.; Tekur, C.; Bhashyam, R.; Nannaka, V.; Mukkamala, R. Using Cuckoo Filters to Improve Performance in Object Store-based Very Large Databases. In Proceedings of the 2023 IEEE 13th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 8–11 March 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 0795–0800.
59. Wang, J.; Gao, Y.; Liu, W.; Wu, W.; Lim, S.J. An asynchronous clustering and mobile data gathering schema based on timer mechanism in wireless sensor networks. *Comput. Mater. Contin* **2019**, *58*, 711–725. [CrossRef]
60. Shafeeq, S.; Zeadally, S.; Alam, M.; Khan, A. Curbing address reuse in the iota distributed ledger: A Cuckoo-filter-based approach. *IEEE Trans. Eng. Manag.* **2019**, *67*, 1244–1255. [CrossRef]
61. Mosharraf, S.I.M.; Adnan, M.A. Improving lookup and query execution performance in distributed Big Data systems using Cuckoo Filter. *J. Big Data* **2022**, *9*, 12. [CrossRef]
62. Wang, J.; Gao, Y.; Yin, X.; Li, F.; Kim, H.J. An enhanced PEGASIS algorithm with mobile sink support for wireless sensor networks. *Wirel. Commun. Mob. Comput.* **2018**, *2018*, 9472075. [CrossRef]
63. Shi, Y.; Li, L.; Yang, J.; Wang, Y.; Hao, S. Center-based Transfer Feature Learning With Classifier Adaptation for surface defect recognition. *Mech. Syst. Signal Process.* **2023**, *188*, 110001. [CrossRef]
64. Ren, Y.; Zhu, F.; Wang, J.; Sharma, P.K.; Ghosh, U. Novel vote scheme for decision-making feedback based on blockchain in internet of vehicles. *IEEE Trans. Intell. Transp. Syst.* **2021**, *23*, 1639–1648. [CrossRef]
65. Soleymani, S.A.; Goudarzi, S.; Anisi, M.H.; Kama, N.; Adli Ismail, S.; Azmi, A.; Zareei, M.; Hanan Abdullah, A. A trust model using edge nodes and a Cuckoo filter for securing VANET under the NLoS condition. *Symmetry* **2020**, *12*, 609. [CrossRef]
66. Wang, Z.; Wang, H.; Wang, Y.; Yang, X. CLASRM: A lightweight and secure certificateless aggregate signature scheme with revocation mechanism for 5G-enabled vehicular networks. *Wirel. Commun. Mob. Comput.* **2022**, *2022*, 3646960. [CrossRef]
67. Xie, Z.; Ding, W.; Wang, H.; Xiao, Y.; Liu, Z. d-Ary Cuckoo Filter: A space efficient data structure for set membership lookup. In Proceedings of the 2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS), Shenzhen, China, 15–17 December 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 190–197.
68. Chen, H.; Liao, L.; Jin, H.; Wu, J. The dynamic Cuckoo filter. In Proceedings of the 2017 IEEE 25th International Conference on Network Protocols (ICNP), Toronto, ON, Canada, 10–13 October 2017; IEEE: Piscataway, NJ, USA, 2017; pp. 1–10.
69. Kwon, M.; Shankar, V.; Reviriego, P. Position-aware Cuckoo filters. In Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems, Ithaca, NY, USA, 23–24 July 2018; pp. 151–153.
70. Reviriego, P.; Martínez, J.; Pontarelli, S. Cfbf: Reducing the insertion time of Cuckoo filters with an integrated bloom filter. *IEEE Commun. Lett.* **2019**, *23*, 1857–1861. [CrossRef]
71. Luo, L.; Guo, D.; Rottenstreich, O.; Ma, R.T.; Luo, X.; Ren, B. The consistent Cuckoo filter. In Proceedings of the IEEE INFOCOM 2019-IEEE Conference on Computer Communications, Paris, France, 29 April–2 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 712–720.
72. Ting, D.; Cole, R. Conditional Cuckoo filters. In Proceedings of the 2021 International Conference on Management of Data, Xi'an, China, 20–25 June 2021; pp. 1838–1850.
73. Oh, H.; Cho, B.; Kim, C.; Park, H.; Seo, J. Anifilter: Parallel and failure-atomic Cuckoo filter for non-volatile memories. In Proceedings of the Fifteenth European Conference on Computer Systems, Heraklion, Greece, 27–30 April 2020; pp. 1–15.
74. Huang, K.; Yang, T. Additive and subtractive Cuckoo filters. In Proceedings of the 2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS), Hang Zhou, China, 15–17 June 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 1–10.
75. Bender, M.A.; Farach-Colton, M.; Johnson, R.; Kuszmaul, B.C.; Medjedovic, D.; Montes, P.; Shetty, P.; Spillane, R.P.; Zadok, E. Don't thrash: How to cache your hash on flash. In Proceedings of the 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11), Portland, OR, USA, 14 June 2011.
76. Wang, M.; Zhou, M. Vacuum filters: More space-efficient and faster replacement for bloom and Cuckoo filters. *Proc. VLDB Endow.* **2019**. [CrossRef]

77. Bawankar, B.; Chinnaiah, K. Implementation of ensemble method on DNA data using various cross validation techniques. *3c Tecnol. Glosas Innov. Apl. Pyme* **2022**, *11*, 59–69. [CrossRef]
78. Breslow, A.D.; Jayasena, N.S. Morton filters: Faster, space-efficient Cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proc. VLDB Endow.* **2018**, *11*, 1041–1055. [CrossRef]
79. Qi, W.; Ovur, S.E.; Li, Z.; Marzullo, A.; Song, R. Multi-sensor guided hand gesture recognition for a teleoperated robot using a recurrent neural network. *IEEE Robot. Autom. Lett.* **2021**, *6*, 6039–6045. [CrossRef]
80. Tian, C.; Xu, Z.; Wang, L.; Liu, Y. Arc fault detection using artificial intelligence: Challenges and benefits. *Math. Biosci. Eng.* **2023**, *20*, 12404–12432. [CrossRef]
81. Qi, W.; Aliverti, A. A multimodal wearable system for continuous and real-time breathing pattern monitoring during daily activity. *IEEE J. Biomed. Health Inform.* **2019**, *24*, 2199–2207. [CrossRef] [PubMed]
82. Liu, Z.; Yang, D.; Wang, Y.; Lu, M.; Li, R. EGNN: Graph structure learning based on evolutionary computation helps more in graph neural networks. *Appl. Soft Comput.* **2023**, *135*, 110040. [CrossRef]
83. Wang, Y.; Liu, Z.; Xu, J.; Yan, W. Heterogeneous Network Representation Learning Approach for Ethereum Identity Identification. *IEEE Trans. Comput. Soc. Syst.* **2022**. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.