Journal of Big Data

RESEARCH Open Access

Improving lookup and query execution performance in distributed Big Data systems using Cuckoo Filter

Sharafat Ibn Mollah Mosharraf[1] and Muhammad Abdullah Adnan[1,2*]

*Correspondence:
adnan@cse.buet.ac.bd
[1] Department of Computer Science & Engineering, Bangladesh University of Engineering & Technology (BUET), Dhaka 1000, Bangladesh
Full list of author information is available at the end of the article

Abstract

Performance is a critical concern when reading and writing data from billions of records stored in a Big Data warehouse. We introduce two scopes for query performance improvement. One is to improve the performance of lookup queries after data deletion in Big Data systems that use Eventual Consistency. We propose a scheme to improve lookup performance after data deletion by using Cuckoo Filter. Another scope for improvement is to avoid unnecessary network round-trips for querying in remote nodes in a distributed Big Data cluster when it is known that the nodes do not have requested partition of data. We propose a scheme using probabilistic filters that are looked up before querying remote nodes so that queries resulting in no data can be skipped from passing through the network. We evaluate our schemes with Cassandra using real dataset and show that each scheme can improve performance of lookup queries for up to 2*x*.

Keywords: Big Data, Distributed systems, Query optimization, Probabilistic data structure, Bloom filter, Cuckoo Filter

Introduction

Big Data systems provide efficient querying of very large amount of data—typically millions to billions of records. While the query performance is better than typical database systems, it is still not very pleasing to end users. For example, Facebook data warehouse has 300 petabytes of data, and a single query processing can take even 350 seconds [1].

Big Data is composed of several components and various researches proposed techniques to improve those components—starting from efficient indexing [2–4] and caching/filtering [5], to techniques such as improved query execution plans [6–8] and effective data partitioning [9–11]. However, we found one aspect of Big Data which existing researches failed to address—improving query execution time by avoiding unnecessary query delegation to remote nodes in a Big Data cluster when it is pre-known that the nodes do not have the data for the requested partition.

Our solution is based on using *Probabilistic Filters*. A probabilistic filter supports set membership queries in such a way that querying a set may result in false positives (claiming an element to be part of the set when it was not inserted), but never in false

negatives (reporting an inserted element to be absent from the set when it is actually present). The most popular probabilistic filter out there is Bloom Filter [5], which is also being used in many popular Big Data systems like Google BigTable [12], Apache HBase [13] and Apache Cassandra [14]. Probabilistic Filter is an interesting software technique that improves query performance by avoiding unnecessary disk accesses, while having extremely low memory footprint requirement and providing extremely fast lookup service. These characteristics also allow it to be passed around among nodes in a network without causing network traffic overhead.

To improve query execution performance in a Big Data cluster, we introduce using a probabilistic filter that will store partition keys against which a node has data within it (we will be referencing this filter as a *Node Filter* throughout this paper). The nodes will synchronize the filters among themselves. A client connects to a node and executes lookup queries (queries that retrieve data) against partition keys, all of which may not be present in the connected node. Before the connected node relays the queries to remote nodes in the cluster that may contain the requested data, it looks up the filter to see if the destination nodes indeed contain the requested data. If not, it can avoid unnecessary network round-trip cost that would have otherwise increased query execution time.

An important limitation of the Bloom Filter is that data cannot be deleted from it [15]. Data deletion is a regular operation in Big Data systems due to cost minimization, privacy issues and effective data analytics [16]. Recently, this limitation of Bloom Filter has been addressed efficiently by another probabilistic filter named *Cuckoo Filter* [15]. Again, Big Data systems that offer high-performance query execution usually utilize Bloom Filters along with *Eventual Consistency* model for performance improvement. Eventual consistency is a consistency model used in distributed computing to achieve high availability that informally guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value. In the Eventual Consistency model, data deleted is not removed from disk storage and rather updated with a deletion marker timestamp called a *Tombstone*. The data is eventually deleted during a data compaction process. This data compaction process takes a lot of CPU resource and has other overheads due to which it is scheduled to occur very infrequently (commonly every few days). This time interval between tombstoning and compaction is called a *grace period*. During the grace period, any lookup query must access disks and retrieve the data only to find out that the data has been deleted. Therefore, while the model improves performance, it fails to do so for the case where data deletion is a regular occurrence.

In this paper, we propose another scheme that improves performance of lookup queries after data deletion by replacing Bloom Filter with Cuckoo Filter that allows deletion of entries from within it. There are a few challenges in implementing this scheme, which we also address properly. We then show that lookup queries after data deletion can improve performance for up to $2x$ using our proposed scheme. We also show that none of our schemes causes performance degradation as a side effect for any other lookup or insertion queries whatsoever.

We have run several carefully-designed experiments in a popular Big Data database (Cassandra) with a real data set to evaluate our schemes and have shown that the performance of lookup queries improves for up to $2x$ in cases where data is deleted or do

not exist in local or remote nodes in a cluster. The experiments cover practical use cases like varying fraction of queries executed in remote nodes and varying fraction of queries returning positive or negative results. We also show that introducing node filters does not cause any performance overhead.

To summarize, in this paper we establish the answer to the question—whether using probabilistic filters improve query execution performance in a networked Big Data cluster. We also explain the significance of this research by referencing renowned and practical cases demonstrating the need to improve query execution performance in a Big Data cluster.

Contributions

The contributions of this paper are as follows:

1. We propose a scheme to improve query execution performance in Distributed Big Data systems.
2. We propose another scheme to improve performance of lookup query after data deletion in Big Data systems that use the *tombstoning* technique for data deletion in an eventual consistency model.
3. We evaluate the proposed schemes using a popular open source Big Data system (Cassandra) and demonstrate that query execution performance improves significantly.
4. We evaluate performance of lookup and insertion queries not covered by our proposed scheme and show that the scheme does not degrade query performance in those other cases as a side effect.

Organization

The rest of this paper is organized as follows. "Background" section provides some background information on Big Data systems and optimizing query execution performance on those systems. "Related works" section provides an overview of researches in concepts related to our scheme. "Proposed Scheme 1: improving performance of lookup queries after data deletion in Big Data systems" and "Proposed Scheme 2: improving performance of lookup in remote nodes in a Big Data clusterookup inRemote Nodes in a Big Data Cluster" sections present our two schemes respectively along with the challenges in implementing those. In "Experimentations" section, we describe our experiment setup with Cassandra, and show the results of evaluations of our proposed schemes and related cases. Finally, we conclude in "Conclusions and future work" section.

Background

Improving query performance in Big Data systems is challenging. There are two primary factors that affect query performance in a distributed Big Data system—huge amount of data to be processed and data transfer among nodes inside a cluster. Naturally, to improve query efficiency, data is partitioned and stored into several nodes. The more nodes need to be accessed to process a single query, the worse the query performance becomes. Consequently, researches to improve query performance in Big Data have

been carried out in three directions—improving query execution performance within a single node, letting less data transfer happen among nodes, and improving query performance by producing efficient query execution plans.

Several researches address the issue of query execution performance within a single node. Accessing storage to retrieve data is the primary bottleneck of query execution. Hence, Big Data systems employ different indexing [2–4], and caching/filtering [5] schemes to improve disk access performance.

To lessen data transfer among nodes, researchers suggest using better data partitioning techniques and schemes. While some researches provide general partitioning technique applicable to any Big Data system with any type of data [9–11], other researches propose schemes to improve query performance for specific categories of data (e.g. geo-spatial [17], locality-aware [18] etc.). It is common to use some form of summary structures to eliminate partitions which are not relevant to a predicate. Most of these techniques improve query performance of a single-server database system (see for example [19] and [20]) and do not apply for distributed database systems. Zigzag Join [21] and Track-join [22] use Bloom Filters with other techniques to minimize data movement over the network for distributed joins. Although these techniques allow eliminating remote data access, they fail to handle the case where they do allow accessing remote data, but the data in reality do not exist in the remote nodes (due to deletion, for example).

We propose a system that can detect absence of data against a partition key from within the node processing the query, thus avoids a network round-trip cost that would otherwise fetch no result for the query. We show that based on the percentage of partition keys that result in no data from different nodes, our scheme can improve query execution performance for up to 100%.

Our proposed scheme basically uses a probabilistic data structure called a *Filter* that provides approximate answer to membership queries. Querying a set for a set membership results in either *true negative* (i.e. reporting an element to be absent from the set) or *false positive* (i.e. claiming an element to be part of the set when it is not). The filter can be queried before reaching a node to know if the node contains records matching the partition key. Thus, a lot of unnecessary network round trip time can be prevented if a probabilistic filter is used. Further information on how probabilistic filter works can be found in [5].

Big Data systems that offer high-performance query execution service usually utilize Bloom Filters along with *Eventual Consistency* model [23] for improving performance of lookup from disk storage. In the Eventual Consistency model, data deleted is not removed from disk storage immediately and rather updated with a deletion marker timestamp called a *Tombstone*. The data is eventually deleted during a data compaction process. This data compaction process takes a lot of CPU resource and has other overheads due to which it is scheduled to occur very infrequently (commonly every few days). This time interval between tombstoning and compaction is called a *grace period*. While the model improves performance, it fails to improve performance for the case where data deletion is a regular occurrence. During the grace period, any lookup query must access disks and retrieve the data only to find out that the data has been deleted. This is due to the fact that Bloom Filter does not support deletion of elements from within it. If, while marking data with tombstones, keys of the marked data could be

removed from Bloom Filter, then this issue would not have caused the performance degradation. To improve performance in this case, we propose a scheme for storage filtering that replaces Bloom Filter with Cuckoo Filter [15]—another probabilistic filter that supports deletion of elements from within it. The scheme tweaks the Cuckoo Filter, too, so that it doesn't cause any side effect when used in a distributed storage environment. We use that modified Cuckoo Filter instead of a Bloom Filter as our chosen storage filter as well as node filter, and show that based on the percentage of partition keys that result in deleted data in remote nodes, our scheme can improve query execution performance for up to 100%.

To summarize, in this paper, we introduce the use of Cuckoo Filters for data lookup in the storage system of a node, as well as for data lookup in remote nodes in distributed Big Data systems. Through experiments, we show that these filters can improve query execution performance for up to 100%.

Related Works

Researches have proposed improvement of various components of Big Data to boost overall query performance. MapReduce is a popular technique that can efficiently query against very large amount of data. A lot of research works have focused on improving the performance of MapReduce [6, 7]. Indexing against large amount of data in Big Data systems is challenging. Researches have been carried out to innovate efficient indexing schemes for Big Data [2–4]. Probabilistic filters have been used to improve query execution performance by avoiding unnecessary disk storage access. Google BigTable [12], Apache HBase [13] and Apache Cassandra [14] are the popular Big Data solutions that utilize Bloom Filter, the most popular probabilistic filter. Several data placement [24, 25] and partitioning techniques [9–11, 17, 18] have been proposed to improve data transfer throughput among nodes in a Big Data cluster. Bloom Filters combined with other techniques are used in distributed joins to minimize data movement over the network (for example, [21] and [22]). However, these techniques fail to handle the case where data in reality do not exist in the remote nodes (due to deletion, for example). We introduce a novel technique of using probabilistic filters to efficiently determine existence of data before relaying query execution to different nodes in a cluster, thereby improving performance for up to 100% in the cases where data do not exist in remote nodes, or data have been deleted from there.

The technique of probabilistic filters was first published by B. H. Bloom [5], which became known as Bloom Filter. While having an extremely small memory footprint, Bloom Filter has the disadvantage of not supporting deletion operation on items within it. Several variations of Bloom Filter have been proposed to address these issues (see [26] for a comprehensive list, and [27] for another recent variation named *Master-Slave Bloom Filter*), but none can address all the issues of Bloom Filter at the same time.

More recently, Bin Fan and others proposed Cuckoo Filter [15] that supports count and deletion operations, does not introduce false negatives during deletion operation, has fewer bits per entry for optimum false positive rate ($< 3\%$), and can maintain stable false positive rate with higher load for up to 95%. After its introduction, recently, several research works have used Cuckoo Filter to speed up the lookup process in the areas of Networking and Security [28–32]. However, there have not been significant researches

utilizing Cuckoo Filter in the area of Big Data—only a few on semi-structured data [33], encrypted data [34] and dynamic data-sets [35].

Cuckoo Filter has a drawback that during insertion, more items may need to be kicked out and placed into alternate buckets, increasing the insertion time. And if exhausted, the filter needs to be resized. Researches have been carried out to devise variations of Cuckoo Filter or Cuckoo Hash that can be used to reduce the insertion time. *Smart-Cuckoo* [36] efficiently predetermines insertion failures without paying a high cost of carrying out step-by-step probing. A. Kirsch et al. [37] showed that the failure probability can be dramatically reduced by the addition of a very small constant-sized *stash*.

In this paper, we utilize the deletability feature of Cuckoo Filter and propose a methodology where querying for data after deletion can improve query execution performance for up to $2x$.

Proposed Scheme 1: improving performance of lookup queries after data deletion in Big Data systems

We first describe the scenario where existing Big Data systems fail to improve query performance. Then we propose our scheme to overcome the limitation. Finally, we state the challenges that we encounter while implementing the proposed scheme and discuss solutions to overcome those.

Query performance degradation in case of data lookup after deletion in existing Big Data systems

Bloom Filter has the limitation that it does not allow deletion of items from within it. Combined with the Eventual Consistency model, this limitation degrades performance significantly for queries looking up data after it has been deleted. The following scenario demonstrates the significance of the issue.

1. When a client deletes a row, instead of actually removing the data from storage, the Big Data database updates the row to add a tombstone marker. The Bloom Filter keeps the row key though, as the key cannot be deleted from the filter.
2. Before the database system runs a storage compaction procedure to actually remove the data (which can be up to a few days later), a client queries with row keys that include the key for the deleted row. Now, Bloom Filter will suggest that the row data exists on disk, as it has the key still stored in it.
3. The database system will read the disk storage only to find out that the row data has been marked with a tombstone and hence removes the row from the resultant set of rows to be returned to the client. This unnecessary storage access increases query execution time significantly and hence degrades query performance.

Proposed scheme to improve performance of lookup query after deletion

We propose a scheme that replaces Bloom Filter with Cuckoo Filter that allows deletion from the filter. Following is an illustrative scenario of how the proposed scheme works and improves performance.

1. A client makes a row deletion query. The query is executed exactly the way executed by the current system, that is, data on disk is marked with a tombstone. We do not propose instant deletion of data and instead propose utilizing the existing tombstoning technique, because data deletion is a very expensive operation that should not be done very frequently.
2. The corresponding row key is deleted from the Cuckoo Filter associated with the row data deleted.
3. The Cuckoo Filter is subsequently flushed into disk to persist the change in it. To make sure the flushing of the filter is handled robustly in a fail-safe way and creates minimal overhead, we propose using *Adaptive Cuckoo Filter* [38], a variant of Cuckoo Filter that removes keys resulting in false positives from itself. "If flushing the updated filter fails after data deletion" section expands upon the issue in detail.
4. Afterwards, during a lookup query, the key will not be found in the filter and hence a costly disk access operation can be avoided, thereby improving performance.

Modifications of Cuckoo Filter to implement proposed scheme

Replacing Bloom Filter with Cuckoo Filter creates new challenges to make the proposed scheme work properly in a fail-safe manner. In this section we discuss the challenges and propose solutions to overcome those.

When consistency level requirement is more than one node

As Eventual Consistency model is a weak consistency model designed to improve performance, Big Data systems employing it usually provide an option to increase confidence in data consistency as may be required by clients. It is worth mentioning here that clients requiring strong consistency should not use Big Data systems that utilize Eventual Consistency; however, there may be cases where a client may not require strong consistency for most of the data, but a particular table/data may be required to maintain greater consistency than the rest.

To specify consistency level requirement, Big Data systems usually provide the option for a query to specify its requirement of consistency level. If there are multiple replicas of a node, a query can ask for higher consistency level of data and the system then executes the same query on multiple nodes and converge the results based on *update timestamp* to make sure the latest data is returned. Now, let us consider the following scenario for a cluster of two nodes (one being a replica of the other) with our proposed method implemented in place.

1. Data is deleted from *Node 1*. The corresponding key is also deleted from the filter. *Node 2* has not been updated with the changes yet.
2. A lookup query is executed with consistency level requirement of 2, that is, two nodes should be compared to find the latest data. The Big Data system will try to converge data from both nodes and see that the former has no data (because the filter says so) while the latter has data, and will conclude that the data in *Node 2* is the latest one, which is clearly wrong. The system derives this conclusion based on comparing *update timestamps* of the data returned from the nodes, and for the former

node, there is no data while the latter one has data with timestamp, which is considered the latest timestamp by the system (Fig. 3).

This can be handled properly by using a modified version of Cuckoo Filter where it will not only respond with whether a key is stored in it or not, but also whether a key in it has been deleted. Thus we can also modify the behavior of the Big Data system to detect this case where consistency level requirement is greater than 1, and let it access the disk to retrieve the row data so that when the timestamps of the data from both the nodes are compared, the system will find the row deletion timestamp to be later than the row creation timestamp and conclude the latest state to be the row deletion, which is the expected result. Following is an illustration of the operations of the modified proposed scheme.

1. A client makes a row deletion query. The query is executed exactly the way executed by the current system, that is, data on disk is marked with a tombstone.
2. The corresponding row key is deleted from the Cuckoo Filter associated with the row data deleted. At the same time, the Cuckoo Filter now maintains a list of deleted keys, and the key is entered into the deleted key list.
3. The Cuckoo Filter is subsequently flushed into disk to persist the change in it. The change in data has not been propagated to the other nodes yet.
4. A lookup query is executed with consistency level requirement of 1, that is, a single node's data will suffice—no convergence of data from multiple nodes needs to be carried out. In this case, the filter will respond that the key is deleted and the database system will avoid the costly disk access operation; resulting in improved query performance.
5. Now, suppose a lookup query is executed with consistency level requirement of 2, that is, two nodes should be compared to find the latest data. The system now detects this and finds the filter responding that the data is deleted. In this case, the system still reads the data and returns the row. The result from two nodes will be converged by the system and as the deleted row will contain the tombstoned timestamp which is the latest timestamp, the system will remove it from the resulting rows of data (Fig. 4).

If flushing the updated filter fails after data deletion

Once a key is deleted from the filter, if flushing the filter into disk storage fails, due to for example crashing of node, then the key will come back into the filter once the node recovers. This will hinder performance improvement of subsequent lookup queries against this key. The following scenario illustrates this point.

1. Data is marked with tombstone and deleted from filter.
2. The filter is scheduled to be flushed into disk.
3. The node crashes before the filter is flushed into disk, and later recovers.

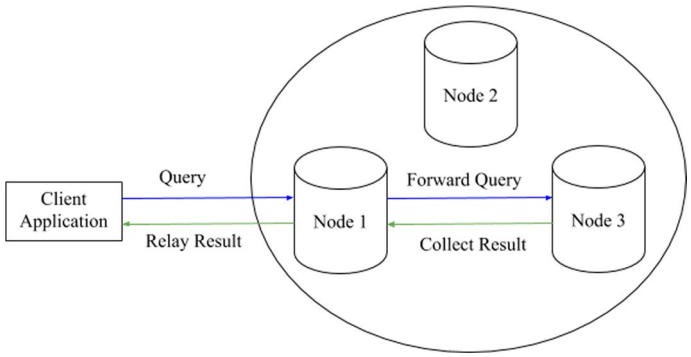

Fig. 1 Query execution path when cluster node *does not own* partition of data requested by client

4. Now the data is tombstoned, but the filter also contains the data. Our proposed method will not gain any performance benefit for that deleted data.

The proper way to ensure reliable updating of filter is using a journaling method to log changes in filter before it is flushed and replaying the log in case the filter fails to get flushed successfully. However, it is a complex process that also has performance overhead.

A rather simpler and more efficient technique would be using the *Adaptive Cuckoo Filter* variant [38], which removes false positives from the filter once detected. So, in this case, once a lookup query finds out that a row key exists in the filter but the row data is tombstoned, it gives the feedback to the *Adaptive Cuckoo Filter* and it removes the row key from within it.

Proposed Scheme 2: improving performance of lookup in remote nodes in a Big Data cluster

Let us first describe a scenario of how a query is executed in a typical Big Data cluster of nodes.

1. A client first connects to a node and executes a query.
2. If the node does not own the partition of data, it fetches the data from the node that owns the partition and relays it to the client (Fig. 1). Note that in this case, if the remote node does not contain any data against the queried partition key, then an unnecessary network round trip occurs, which increases query execution time.
3. If the node, that has been connected to, owns the partition of data queried by the client, it executes the query and returns the result to the client (Fig. 2). In this case, a storage filter is looked up to see if the data actually exist on the disk storage. If the storage filter confirms that the data do not exist on the disk storage, then a disk lookup cost can be avoided. However, when Bloom Filter is used as storage filter, for case of deleted data, it replies with a false positive that the data exist on disk storage. It thereby increases query execution time.

Now let us describe our scheme and show how it fits into the above-mentioned scenario and improves performance.

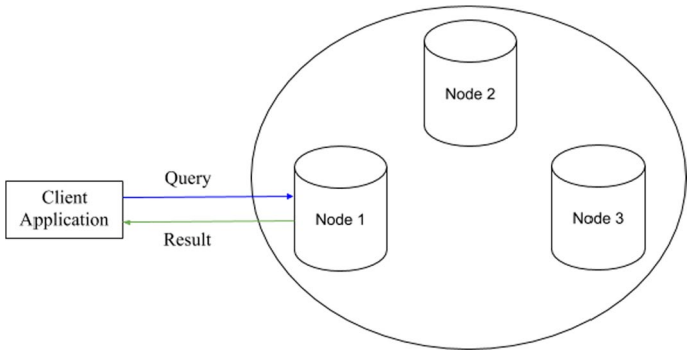

Fig. 2 Query execution path when cluster node *owns* partition of data requested by client

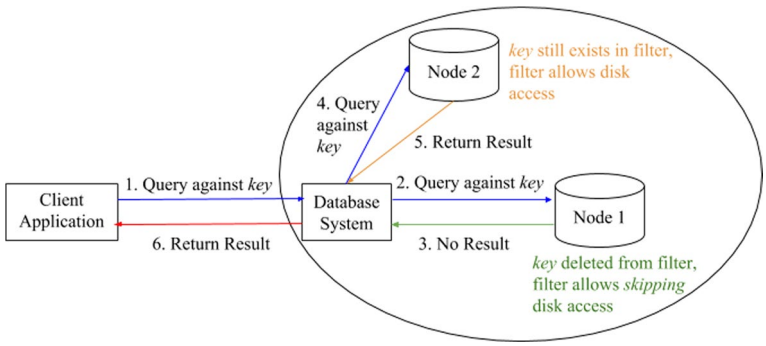

Fig. 3 Failure of Database System to infer correct result with our proposed scheme applied, in the case where Consistency Level requirement > 1

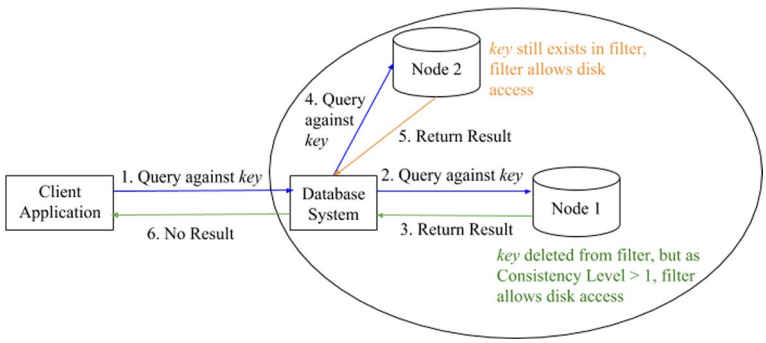

Fig. 4 Modifying the behavior of Cuckoo Filter makes sure the Database System infers correct result with our proposed scheme applied, in the case where Consistency Level requirement > 1

1. Each node in a cluster of node maintains a filter (we call it a *Node Filter*) against each table in the database. A node filter contains all the keys against which data is stored on that node. The node filter is synchronized among all the nodes in the cluster, so that each node has a copy of node filters of all other nodes in the cluster.
2. When a client needs to look up data from a Big Data system, it first connects to a node in a cluster of nodes and executes a lookup query against a set of keys.

3. For each key, the database system looks up whether data against it is stored in that node, or in some remote nodes in the cluster.
4. If the data against a key is stored in that node, the database system queries the storage filter to see if the data really exist on the local storage. If the filter replies in affirmative the database system looks up the storage to retrieve the data. If the filter replies in negative, it then knows that data against the key cannot be found.
5. If the data against a key is not stored in that node, the database system identifies the remote node to look up. It then checks the node filter of that remote node to see if data really exist on that node. If the filter replies in affirmative, the database system delegates the query to that node to retrieve the data from it. If the filter replies in negative, it then knows that data against the key cannot be found in the remote node, and thus save a network round-trip latency.

Figure 18 shows the system architecture as a whole.

Using Cuckoo Filter to improve lookup performance after data deletion

Step 1 in the above proposed scheme states that any change against a key in the data stored in disk would trigger corresponding change to the key stored in the Node Filter. This is always true for data insertion. However, not in the case of data deletion. Most databases employ the *Eventual Consistency* model to improve read/write performance. Due to constraints of that model, data deletion is not executed immediately. Rather data to be deleted is updated with a deletion timestamp called a *Tombstone*. In regular intervals, data is compacted and at that time tombstoned data is actually removed. The interval typically is set to a few days due to the performance impact of the compaction process. Hence, queries executed after tombstoning but before compaction need to retrieve all the data from storage, then remove the tombstoned data from the resultant row set. As Bloom Filter cannot delete elements from within it, the tombstoned keys cannot be deleted from the node filter; thus Bloom Filter cannot improve performance in this case. However, as Cuckoo Filter supports data deletion, tombstoned keys can be removed from the node filter, allowing to skip queries from being executed in the remote nodes and improve query execution performance thereby. It is for this reason we propose using Cuckoo Filter so that the performance benefit of our scheme becomes applicable in case of data deletion changes, too. We compare the performances of using Bloom Filter versus Cuckoo Filter in case of data lookup after deletion and show that Cuckoo Filter can improve query execution performance for up to 100% in this case.

Challenges and issues

Obviously introducing a level of filtering comes with its own overheads and challenges. In this section we address the challenges associated with Node Filter and Storage Filter and propose effective solutions.

Synchronizing node filters among nodes

The most effective way to synchronize node filters among nodes is whenever a node filter changes. However, a node can become unresponsive due to crashing for example, and

hence it will fail to receive the changed node filters from other nodes. Then when that unresponsive node recovers, it now has an outdated version of node filters; and if node filters do not change further, that node will always contain an outdated node filter (Fig. 19). The usual way to address this issue is to implement an agent that will take care of detecting node crashing and recovery and synchronize node filters accordingly. The implementation of this solution is different for distributed master-client architecture (Fig. 20) and peer-to-peer architecture (Fig. 21). However, this solution is complex and has its own overhead.

Instead, we propose to synchronize node filters periodically. This is based on the fact that the size of the Filter is extremely small (only 1 megabyte for having 1 million entries, assuming false-positive probability is 0.01%), and hence network data transfer overhead to synchronize the filters will not degrade overall network performance significantly. The interval of synchronization can be set based on number of stored partition keys. For example, if there are 1M keys stored in a node partition, then node filter size will be pretty small and we can set the synchronization interval to 5 minutes only. If, however, there are 1B keys stored in a node partition, then node filter size will be around 1GB and so we can set the synchronization interval to 1 hour. Note that having 1B unique partition keys is not practical as it will degrade query performance heavily. In any case, if filter size becomes an issue, we can always tune the false positive probability of the filters to make the filter size significantly lower.

False-negatives within the time duration of filter update and synchronization

If we choose to synchronize filters among nodes periodically, then there is a corner-case that may be unwelcoming. Consider the scenario when data with new partition keys are inserted into a node, but the next interval for node filter synchronization is yet to come. At that moment, a client connects to another node and executes a query that needs to look up from the newly inserted data. Then the connected node will not reach the node having the new data because the node filter in the connected node will respond that partition keys for those newly inserted data do not exist (Fig. 22).

This is a common scenario in Big Data systems that provide high performance and availability. That is why most Big Data systems choose to employ Eventual Consistency, which is a form of weak consistency. Similar to expectations from a weak consistency system, the filter synchronization can also be considered as a weak consistency one. It should be emphasized here that due to the extremely small size of the filters, filter synchronization interval can be set to as low as every few seconds, without causing increase in network traffic. In case the filter size becomes large, false positive rate of the filter can be sacrificed for lowering filter size, and that would still improve query execution performance significantly. Finally, in case of strong consistency requirement, the database system or queries can be configured to probe data from multiple replicas and converge the results into the most consistent result set. That will in any case make sure of producing consistent data irrespective of using node filters.

Experimentations

Experiments setup

Most Big Data systems currently in use employ Eventual Consistency as well as Bloom Filters. Eventual Consistency is a popular choice because it allows faster read/write

operations, and Bloom Filter plays a vital role in improving query execution performance by avoiding disk access in the case where data do not exist on disk. Examples of popular Big Data systems incorporating these features include Google BigTable [12], Apache HBase [13] and Apache Cassandra [14]. Among these, we choose Cassandra to be our experimental Big Data system. The driving factors for choosing Cassandra are as follows.

1. Apache Cassandra is a free and open-source distributed Wide-Column-Store NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.[1] Cassandra offers robust support for clusters spanning multiple datacenters [39] with asynchronous masterless replication allowing low latency operations for all clients
2. It is the most popular Column-Store Big Data database as of Jan. 2020 according to DB-Engines ranking.[2]
3. The source code of Cassandra is organized and clean. Also, querying the database is very simple syntactically and programmatically compared to other Big Data databases, along with having performance tracing feature built-in.

We forked the Cassandra source code repository into our own repository on Github[3] and plugged an open-source implementation of Cuckoo Filter[4] into it. We made necessary changes according to our proposed methodology.

We compiled the source code and ran our experiments in a Cassandra Cluster set up in Amazon Web Services (AWS).

As for the experimentation of our proposed scheme for improving performance of lookup queries after data deletion, we set up Cassandra on a single AWS node that was an m5.xlarge EC2 instance—2.5 GHz Intel Xeon Platinum 8175 processor with 4 virtual CPUs, 16 GB RAM and 200 GB io1 EBS volume with 10,000 IOPS, having up to 3500 Mbps dedicated EBS bandwidth.

As for the experimentation of our other proposed scheme for improving performance of lookup in remote nodes, we set up a Cassandra cluster on 6 AWS nodes. The 6 nodes in the cluster was spread in three availability zones (us-east-2a, us-east-2b, us-east-2c) in the US-East (Ohio) region. Each node had the same configuration as the one used for experimenting the first scheme. The experiments were executed from a same-configuration node in one of the availability zones of the cluster (us-east-2a).

As for a real-world experiment dataset, we chose to use Amazon Customer Reviews dataset (~136M records, ~50GB total data size) that is available publicly.[5] We inserted the dataset into our Cassandra instance using the built-in *CQLSH* tool for data manipulation.

[1] https://en.wikipedia.org/wiki/Apache_Cassandra.

[2] https://db-engines.com/en/ranking/wide+column+store.

[3] https://github.com/sharafat/cassandra.

[4] https://github.com/MGunlogson/CuckooFilter4J.

[5] https://registry.opendata.aws/amazon-reviews.

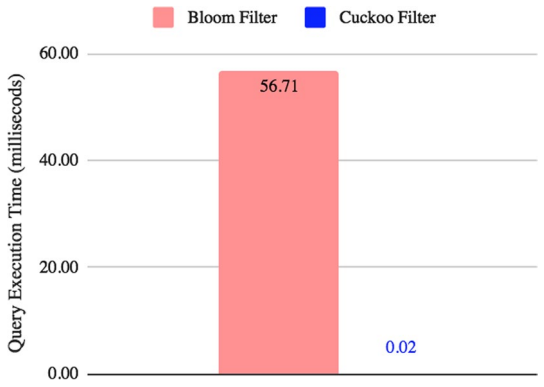

Fig. 5 Lookup performance after deletion (lower is better). Cuckoo Filter improves performance for up to 100%

We developed a program in Java[6] to connect to our Cassandra instance, execute queries and collect execution time.

Experiments setup and execution details

This section outlines the steps of setting up and running the experiment.

First, a cluster of our modified version of the Apache Cassandra system needs to be set up. Each node should be set up on different physical machines. The Cassandra system on the master node should be configured so that it can identify the rest of the nodes in the cluster. Then, the experiment dataset should be loaded on the master node. Due to cluster configuration, the dataset would get replicated to all the nodes in the cluster. This completes the experiment setup. Afterwards, the Cassandra Query Language Shell binary file (`cqlsh`) is used to execute the various selection queries and measure performance of the query execution.

Experimental results 1: performance of lookup queries after data deletion

In "Lookup performance after deletion" section, we evaluate the performance of our proposed scheme to improve performance of lookup query after data deletion in an eventually consistent database system. To show that the core change proposed in our scheme, that is, replacing the Bloom Filter with Cuckoo Filter, does not degrade performance of Cassandra, we measure lookup and insertion query performances in general and present the result of our evaluation in "Lookup performance in general" and "Insertion performance" sections.

Lookup performance after deletion

From Table 1 and Fig. 5 we see that allowing deletion in filter improves query execution performance significantly (in this particular instance for 99.96%) for querying data that have been deleted. This leads to the conclusion that the more query results in empty result, the faster the query executes. We have also done an experiment that

[6] https://github.com/sharafat/cuckoo-filter-performance-analyzer.

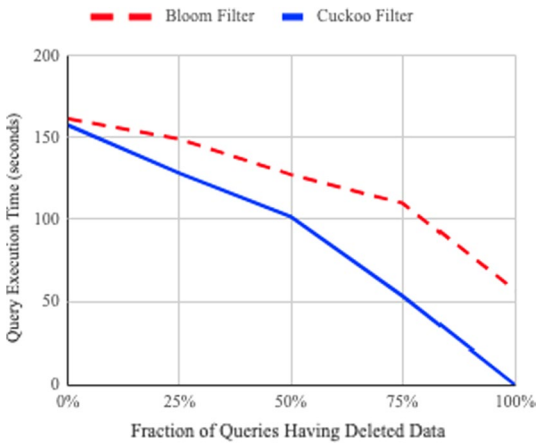

Fig. 6 Lookup performance after deletion for varying percentage of queries returning deleted data (lower is better). Cuckoo Filter improves performance for up to 100%

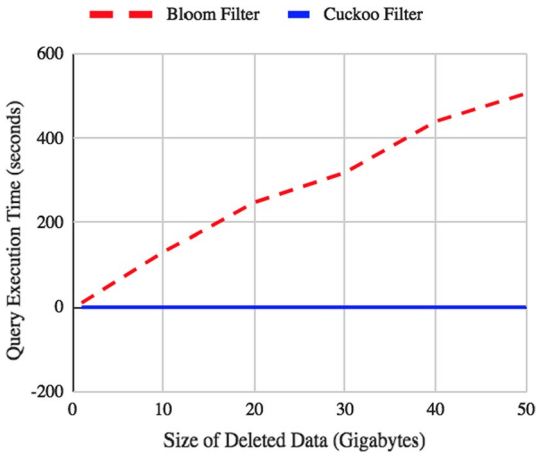

Fig. 7 Lookup performance after deletion for varying data size (lower is better). Cuckoo Filter improves performance for up to 100%

proves this conclusion is correct. Table 2 and Fig. 6 shows the query execution times where fraction of queries look up keys that have their data deleted. We can see that when none of the queries contain keys that have been deleted, that is, fraction of deleted data queries is 0%, the performance is almost identical. The more the fraction of queries look up keys having their data deleted, the better performance becomes; up to the point where query execution time comes down to few milliseconds from many seconds, yielding almost 100% ($2x$) performance gain.

It should be noted here that in case of Bloom Filter, along with increase in fraction of deleted data queries, the query execution time decreases a bit. That is because the query execution time includes the data transfer time from the server to the client, and hence the lesser the data for transfer, the lesser query execution time becomes.

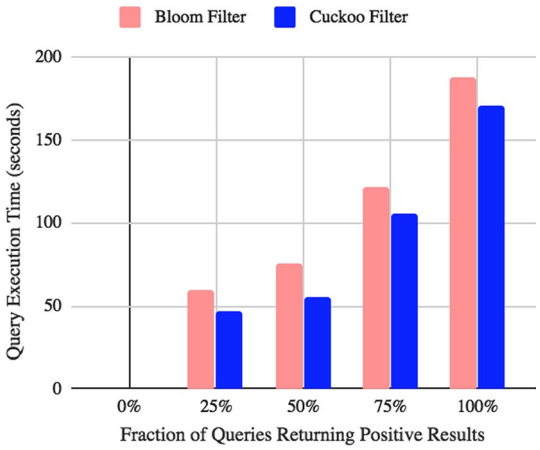

Fig. 8 Lookup performance for varying query result positiveness (that is, percentage of queries returning rows, as opposed to returning 0 rows) (lower is better). Performance of Cuckoo Filter is not worse than Bloom Filter

Table 1 Experimental results 1: lookup performance after deletion. Query execution time in milliseconds

Bloom filter	Cuckoo Filter	Improvement
56.71	0.02	99.96%

Table 2 Experimental results 1: Lookup performance after deletion for varying queries retrieving deleted rows. Query execution time in seconds. (Each query results in 5M rows)

Deleted key fraction (%)	Bloom Filter	Cuckoo Filter	Improvement (%)
0	161.368	157.392	2.46
25	148.943	128.170	13.95
50	127.332	101.672	20.15
75	109.944	53.448	51.39
100	56.709	0.004	99.99

Table 3 Experimental results 1: Lookup performance after deletion for varying data size. Query execution time in seconds

Data size (GB)	Bloom filter	Cuckoo Filter	Improvement (%)
1	10.258	0.005	99.95
10	130.664	0.002	100.00
20	248.105	0.006	100.00
30	318.843	0.009	100.00
40	440.009	0.014	100.00
50	506.697	0.020	100.00

We also experimented with lookup after deletion performance against varying data size, and from Table 3 and Fig. 7 it can be seen that the more data is deleted, the more performance improvement Cuckoo Filter achieves.

Table 4 Experimental results 1: Lookup performance varying the fraction of queries yielding positive results. Query execution time in seconds

Positive result fraction (%)	Bloom filter	Cuckoo Filter	Improvement (%)
0	0.025	0.021	16.00
25	59.904	47.155	21.28
50	75.618	54.873	27.43
75	121.482	105.444	13.20
100	187.531	170.694	8.98

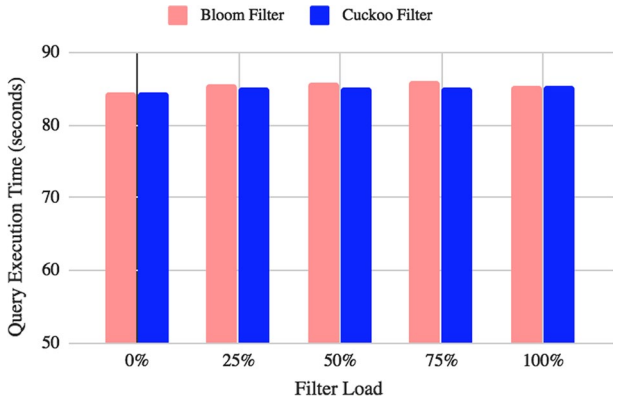

Fig. 9 Insertion performance for varying filter load (that is, how full the filter is) (lower is better). Performance of Cuckoo Filter is not worse than Bloom Filter

Lookup performance in general

Table 4 and Fig. 8 shows the comparison result for varying the fraction of queries that return positive results, that is, return rows as opposed to resulting in 0 rows. Naturally, the more queries return positive results, the more data need to be transferred from server to client, hence the query execution time increases. But the query execution time for both the cases of Bloom Filter and Cuckoo Filter remains very similar, with Cuckoo Filter not causing performance degradation of lookup queries in general.

Insertion performance

Table 5 and Fig. 9 shows the result of our experiment with insertion query execution (~ 136M rows) and it clearly shows that Cuckoo Filter performance is again equivalent to Bloom Filter, while being slightly better.

Experimental results 2: performance of lookup in remote nodes

In "Lookup performance after deletion" section, we evaluate the performance of our proposed system to improve lookup performance in a Big Data cluster by knowing beforehand if a remote node actually contains a requested partition of data. The evaluation is conducted by comparing the default implementation to Node Filters utilizing Bloom and Cuckoo Filters. We show in "Lookup performance in general" section that Cuckoo Filter dramatically improves performance over Bloom Filter in case of

Table 5 Experimental results 1: Insertion performance. Query execution time in seconds

Filter load (%)	Bloom filter	Cuckoo Filter	Improvement (%)
0	84.565	84.457	0.13
25	85.572	85.224	0.41
50	85.863	85.183	0.79
75	86.029	85.255	0.90
100	85.435	85.413	0.03

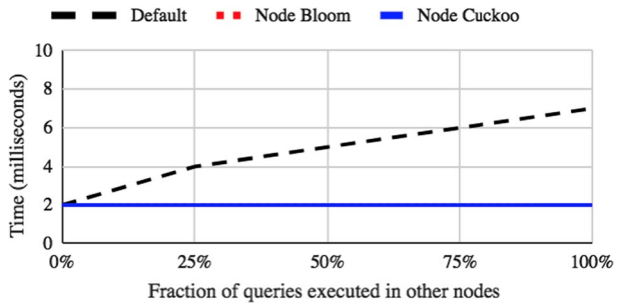

Fig. 10 Lookup performance by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns negative results (lower is better). The performances of Node Bloom and Node Cuckoo are very similar and hence overlap

lookup query after data deletion. In "Lookup performance when remote nodes do not contain data against queried partition key" section, we evaluate the performance of our proposed system when 100% queries result in positive data and conclude that the proposed system does not degrade performance. Again, in "Insertion performance" section, we evaluate the performance of the system in case of data insertion and show that it does not degrade performance. Finally, section compares the CPU and network bandwidth utilization of our proposed system to that of the default system, and yet again we show that the resource utilization overhead of our proposed system is negligible.

Lookup performance when remote nodes do not contain data against queried partition key

We first evaluate the performance of our proposed method by measuring query execution time against varying a fraction of queries to be executed in the remote nodes. We maintain that none of the queries that are executed in the connected node or the remote nodes results in any row. Table 6 and Fig. 10 shows the result and from it we can see that while the performance of the default implementation degrades along with increased number of queries to be executed in remote node, the node filter implementation improves performance for up to around 70% (1.7x).

Lookup performance after data deletion

Due to the constraints of Eventual Consistency model used in most distributed Big Data systems, data is actually not deleted for a long time (typically few days), but rather

Table 6 Experimental results 2: lookup performance when remote nodes do not contain data against queried partition key. Query execution time in milliseconds

Fraction of queries executing in connected vs other nodes, while all keys returning negative results (%)	Default Implementation	Node Bloom	Node Cuckoo	Improvement (%)
0	2	2	2	0
25	4	2	2	50
50	5	2	2	60
75	6	2	2	67
100	7	2	2	71

Table 7 Experimental results 2: lookup performance after data deletion. Query execution time in seconds

Fraction of queries executing in connected vs other nodes, while all keys returning deleted results (%)	Default Implementation	Node Bloom	Node Cuckoo	Improvement (%)
0	81.333	81.26	81.67	0
25	92.879	91.60	82.91	11
50	94.189	94.81	58.10	38
75	100.382	100.83	34.31	66
100	87.539	86.29	0.00	100

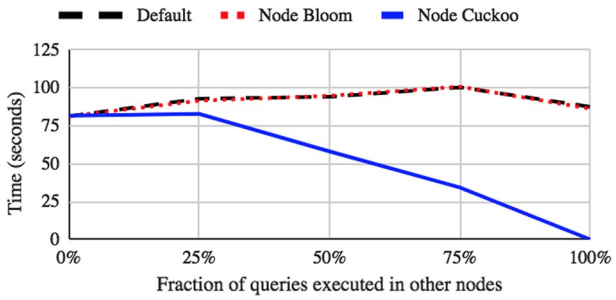

Fig. 11 Lookup performance after data deletion, by varying fraction of queries retrieving rows from remote node (lower is better). All the queries result in deleted data. Node Cuckoo Filter improves lookup performance for up to 100%, depending on the fraction of queries executed in other nodes. The performances of the default implementation and Node Bloom are very similar and hence overlap

marked with a deletion marker called *Tombstone*. As Bloom Filter does not support deletion of items from within it, the Node Bloom Filter allows querying the remote node which returns empty result upon detecting the tombstones. Cuckoo Filter supports deletion of items from within it and in this case, it can prevent the unnecessary round trip. Though the performance of Node Cuckoo Filter and Node Bloom Filter is similar, in this case, Node Cuckoo Filter outperforms Node Bloom Filter for up to 100% ($2x$), as can be seen from Table 7 and Fig. 11.

It is worth mentioning here that when all queries are executed in remote nodes, the database system can save time as no further processing is required for retrieving rows from the local storage of the node. That is why in Table 7 and Fig. 11 the query

Table 8 Experimental Results 2: lookup performance after data deletion by varying data size. Query execution time in seconds

Data size (GB)	Default Implementation	Node Bloom	Node Cuckoo	Improvement (%)
1	0.007	0.002	0.001	50
10	114.364	111.930	0.001	100
20	184.581	183.720	0.002	100
30	239.622	240.370	0.001	100
40	322.622	321.272	0.002	100

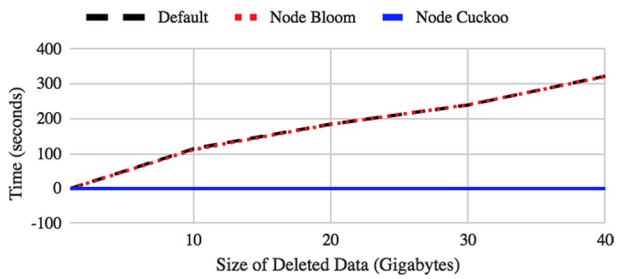

Fig. 12 Lookup performance after data deletion, by varying deleted data size (lower is better). All the queries are relayed to remote nodes for execution. Node Cuckoo Filter improves lookup performance for up to 100%

execution time of the default implementation as well as Node Bloom takes less time when 100% of the queries are executed locally or remotely, compared to when some of the queries are executed locally and some remotely.

We also experimented with lookup after deletion performance against varying data size, and from Table 8 and Fig. 12 it can be seen that the more data is deleted, the more performance improvement Node Cuckoo Filter achieves.

Lookup performance when remote nodes contain data against all queried partition keys

We now evaluate the performance of our proposed method by measuring query execution time against varying a fraction of queries to be executed in remote nodes. We maintain that all of the queries that are executed in the connected node or the remote nodes result in some rows. Table 9 and Fig. 13 shows the result of the experiment that the performance is very similar for all implementations. Hence, we conclude that introducing node filter does not significantly degrade performance of lookup queries that return positive results.

Table 9 Experimental results 2: lookup performance when remote nodes contain data against all queried partition keys. Query execution time in seconds

Fraction of queries executing in connected vs other nodes, while all keys returning positive results (%)	Default Implementation	Node Bloom	Node Cuckoo	Improvement (%)
0	200.290	200.852	200.737	0
25	236.479	237.785	237.603	0
50	248.119	249.285	248.927	−1
75	276.972	277.024	277.042	0
100	254.769	255.881	255.497	0

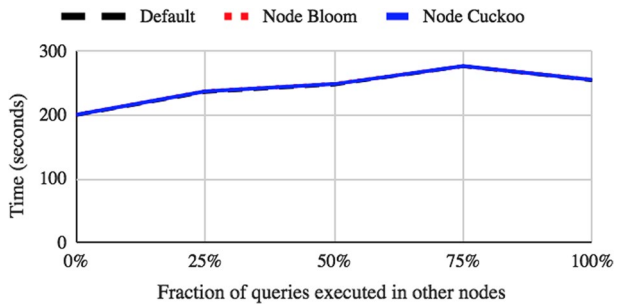

Fig. 13 Lookup performance by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns positive results (lower is better). The performances of all the implementations are very similar and hence overlap

Table 10 Experimental Results 2: insertion performance. Query execution time in minutes

Default implementation	Node bloom	Node Cuckoo
78.27	78.03	77.94

Insertion performance

Table 10 and Fig. 14 shows the performance comparison of data insertion (6 nodes and ~136M rows). The performance of all implementations are almost the same. Therefore, our proposed system does not degrade performance in case of insertion queries either.

Resource utilization comparison

Our proposed system has two overheads compared to the default implementation—(1) during data insertion and deletion, node filter needs to be created/updated; and (2) the node filters need to be synced among nodes.

While running the experiments, we also measured the CPU utilization and network bandwidth utilization of both the default and our proposed system using AWS Cloud-Watch Monitor.

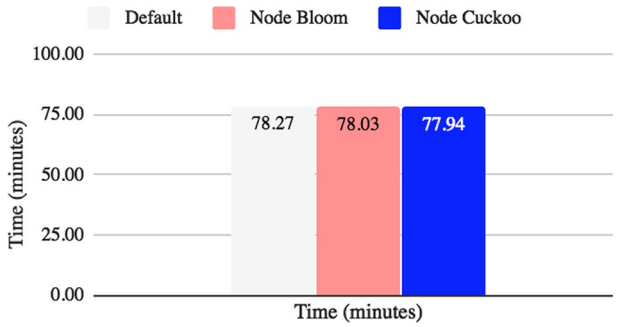

Fig. 14 Insertion Performance (lower is better). The performances of all the implementations are very similar

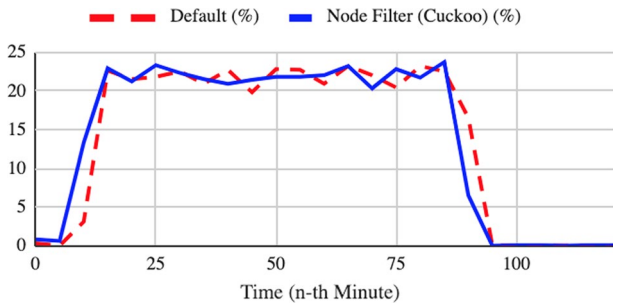

Fig. 15 CPU utilization (lower is better). The CPU overhead of the proposed solution is negligible

Table 11 CPU utilization by Node Filter

Time (*n-th* Minute)	Default implementation (%)	Node Filter (Cuckoo) (%)
0	0.3	0.8
5	0.0	0.6
10	3.1	13.3
15	22.6	22.9
20	21.5	21.2
25	21.8	23.3
30	22.5	22.3
35	20.9	21.5
40	22.7	20.9
45	19.8	21.4
50	22.8	21.8
55	22.7	21.8
60	20.9	22.0
65	23.2	23.2
70	22.5	20.3
75	16.5	22.8
80	23.2	21.7
85	22.5	23.7
90	16.5	6.5
95	0.0	0.0
100	0.0	0.1
105	0.0	0.1
110	0.0	0.0
115	0.0	0.1
120	0.0	0.1

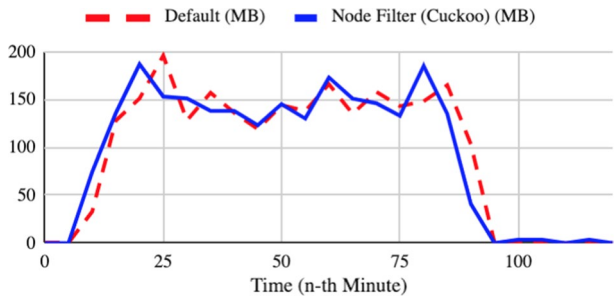

Fig. 16 Network bandwidth consumption (incoming) in megabytes (lower is better). The overhead of the proposed solution is negligible

Table 12 Network bandwidth consumption (incoming) in megabytes by Node Filter

Time (*n-th* Minute)	Default Implementation (MB)	Node Filter (Cuckoo) (MB)
0	0.6	0.0
5	0.5	0.1
10	33.1	74.6
15	128.0	136.0
20	151.0	187.0
25	196.0	153.0
30	128.0	151.0
35	157.0	138.0
40	136.0	138.0
45	119.0	123.0
50	144.0	145.0
55	138.0	130.0
60	166.0	173.0
65	135.0	151.0
70	158.0	146.0
75	143.0	133.0
80	148.0	185.0
85	165.0	135.0
90	103.0	40.9
95	0.1	0.1
100	0.1	3.5
105	0.1	3.5
110	0.1	0.1
115	0.0	3.5
120	0.0	0.1

Table 11 and Fig. 15 show the CPU utilization comparison. We can see that our proposed solution does not cause any significant overhead. Even during the node filter synchronization, the CPU overhead of our proposed system is zero.

Table 12 and Fig. 16 show the Network bandwidth consumed by incoming data; whereas Tables 13 and 17 show the Network bandwidth consumed by outgoing data. From the figures it can be seen that again, the network bandwidth overhead is negligible. During the node filter synchronization interval, the bandwidth overhead is equal

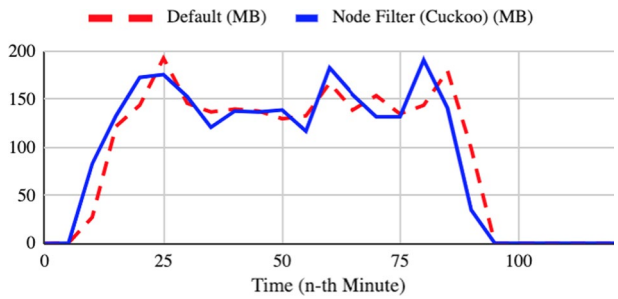

Fig. 17 Network bandwidth consumption (outgoing) in megabytes (lower is better). The overhead of the proposed solution is negligible

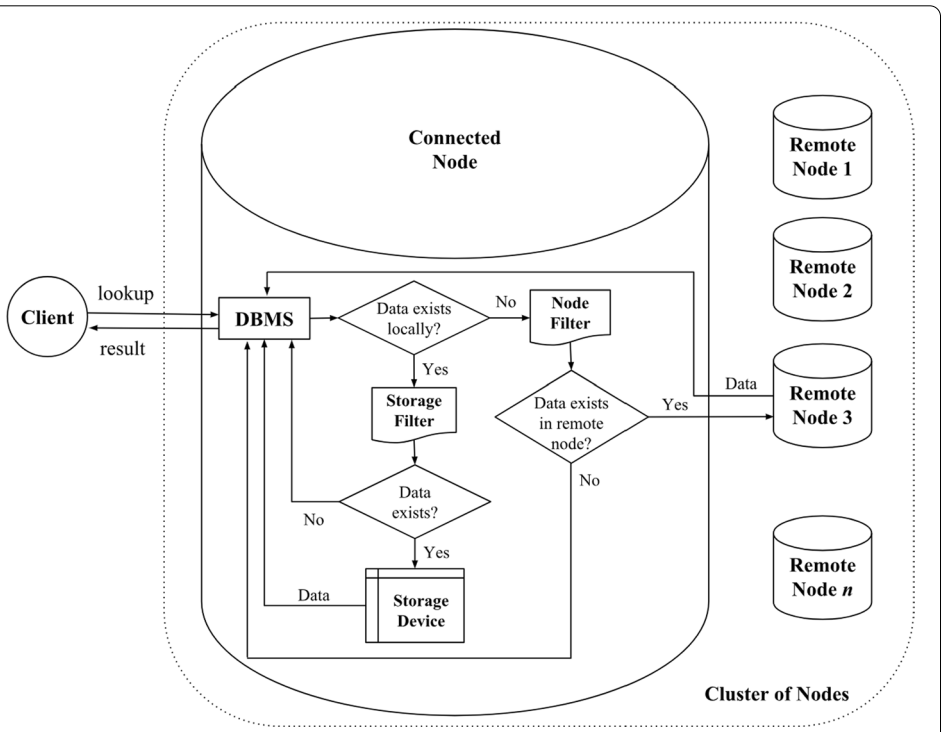

Fig. 18 System Architecture to lookup data in a Big Data cluster

to the node filter size; which, in our experiment was only around 3.55 megabytes, as is evident from Fig. 16 (during 100th minute and onwards). For most practical purposes, the network bandwidth overhead will be negligible.

Conclusions and future work

Improving query performance is one of the most challenging issues in Big Data. Several techniques have been utilized by various Big Data systems to improve query performance as much as possible, including employing the Eventual Consistency Model and Bloom Filters. Bloom Filter degrades performance of lookup queries after data deletion in an Eventual Consistent database. Also, none of the techniques address

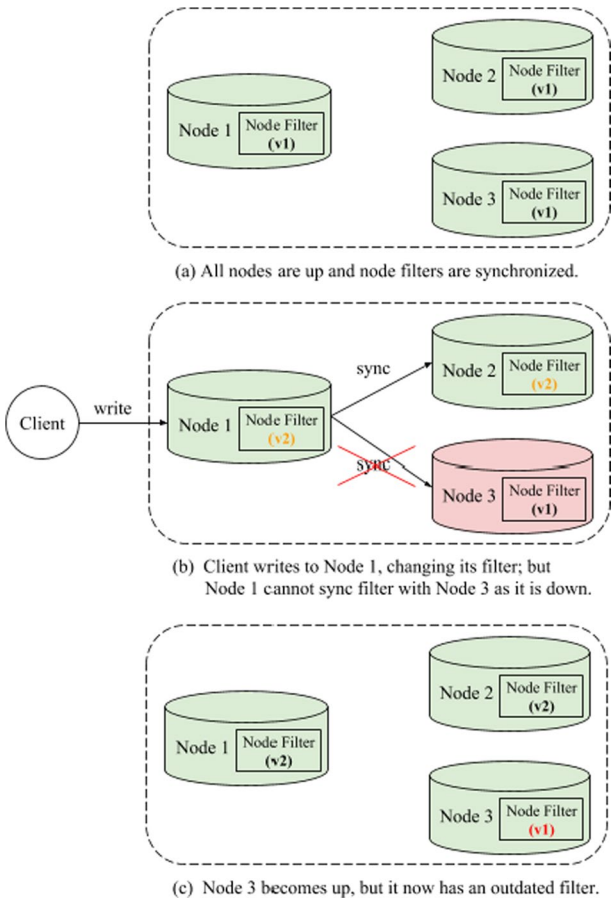

Fig. 19 Node filter synchronization issue during failure of a node

the issue of performance improvement by having pre-existing knowledge of partition data existence in each node of a Big Data cluster.

This paper proposes two schemes that improve query performance significantly. The first scheme improves performance of lookup queries after data deletion in an Eventually Consistent Big Data system by replacing Bloom Filter with a modified version of Cuckoo Filter. The other scheme improves performance of lookup in remote nodes in a Big Data cluster by placing and synchronizing a probabilistic filter with each node and looking it up to see if data exist in remote nodes before relaying queries to those directly. We also recommend using Cuckoo Filters instead of Bloom Filters in the proposed system, that will result in the added benefit of improved performance of lookup queries after data deletion.

Both the schemes have been evaluated with a popular Big Data database (Cassandra) with a real dataset. The first scheme has been shown to improve performance of lookup queries after data deletion for up to 100% ($2x$) in an Eventual Consistent database. This is in consistent with the expectation based on previous researches that Cuckoo Filter outperforms Bloom Filter in cases where data is deleted in regular interval. The other scheme has been shown to improve performance of lookup queries for up to 100% ($2x$) in cases where data do not exist in remote nodes, or have been deleted there (in case

Table 13 Network bandwidth consumption (outgoing) in megabytes by Node Filter

Time (*n*-th minute)	Default implementation (MB)	Node filter (Cuckoo) (MB)
0	0.1	0.0
5	0.5	0.2
10	27.4	83.1
15	122.0	133.0
20	144.0	173.0
25	193.0	176.0
30	146.0	153.0
35	137.0	121.0
40	140.0	138.0
45	138.0	137.0
50	130.0	139.0
55	133.0	117.0
60	167.0	183.0
65	139.0	155.0
70	154.0	132.0
75	135.0	132.0
80	144.0	191.0
85	180.0	141.0
90	99.0	34.8
95	0.1	0.1
100	0.1	0.1
105	0.1	0.1
110	0.1	0.1
115	0.0	0.1
120	0.0	0.1

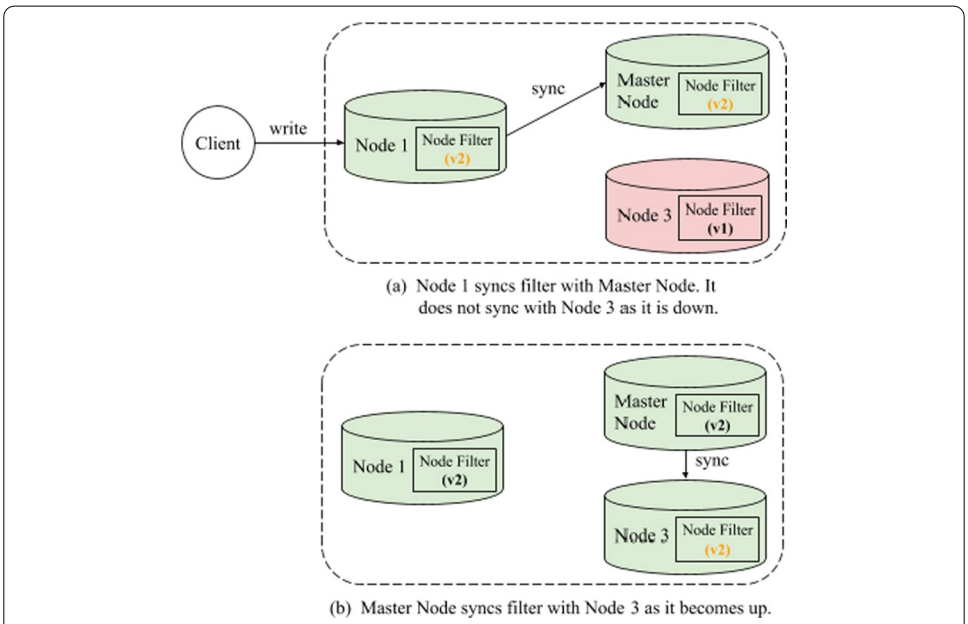

Fig. 20 Proper way to synchronize node filter in master-client distributed architecture

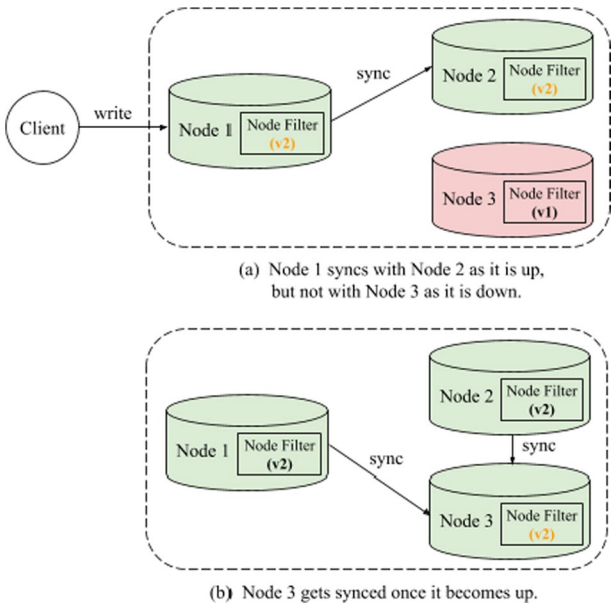

Fig. 21 Proper way to synchronize node filter in peer-to-peer distributed architecture

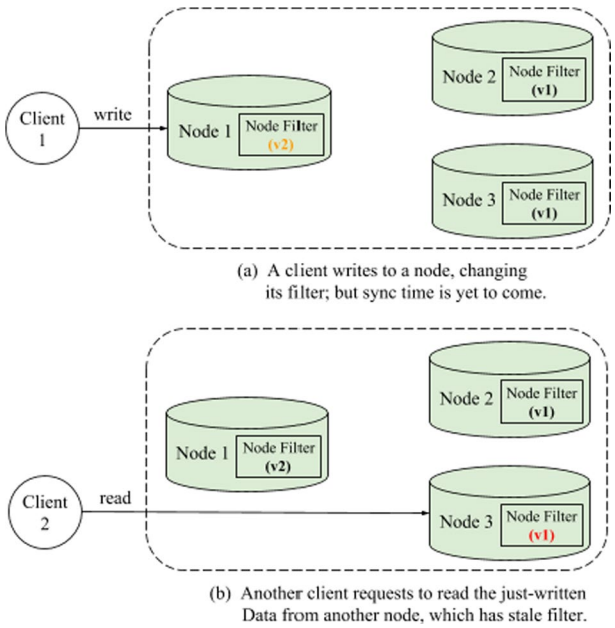

Fig. 22 False-negatives within the time between filter update and synchronization

of Eventual Consistent databases). The experimental results for this scheme prove our hypothesis that using a Probabilistic Filter can improve query execution performance significantly in distributed Big Data clusters. We believe that utilizing the most popular column-store Big Data system (namely Cassandra) to prove our hypothesis would encourage researchers and Database System experts to further experiment with Probabilistic Filters to improve query performance in distributed Big Data systems.

Our proposed scheme improves query performance significantly when data deletion is performed regularly. There are various practical needs for regular data deletion. One is that unnecessary data costs storage a lot. For example, FTI Consulting, the renowned business advisory firm revealed that it worked with a bank to help delete hundreds of terabytes of useless data, and in the process the company saved over $3 million over 5 years [16]. Apart from storage costs, cloud providers offering Big Data services charge for query executions based on how much data the query needs to process to return result. Therefore, the more data is there, the costlier each query execution becomes; and hence deleting unnecessary data would cut down costs at a great extent. These practical use-cases show the necessity of deleting/cleaning data regularly, which implies that using our proposed schemes, distributed Big Data systems can save not only time, but also money.

In future, we would like to analyze the false positive rate of node filters, as the less false positive rate exhibited by probabilistic filters, the better performance improvement becomes. We will also explore and evaluate other filtering techniques that may improve query performance further. Moreover, we plan to explore the consistency and availability models used by Big Data systems to investigate cases where query performance may be hampered and come up with schemes to address those limitations.

Acknowledgements
This is an extended version of the conference paper by the same authors published in IEEE Big Data 2018 titled "Improving Query Execution Performance in Big Data using Cuckoo Filter".

Authors' contributions
Both authors brainstormed, researched and analyzed the research topic and proposed schemes. SIMM performed the experiments necessary for the proposed schemes, and was a major contributor in writing the manuscript. Both authors read and approved the final manuscript.

Funding
Not applicable.

Availability of data and materials
The datasets generated and/or analyzed during the current study are available in the Amazon Customer Reviews dataset that is available publicly at https://registry.opendata.aws/amazon-reviews.

Declarations

Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
[1]Department of Computer Science & Engineering, Bangladesh University of Engineering & Technology (BUET), Dhaka 1000, Bangladesh. [2]Department of Computer Science & Engineering, University of California San Diego, California, USA.

Received: 4 September 2021 Accepted: 10 January 2022
Published online: 26 January 2022

References

1. Vagata P, Wilfong K. Scaling the Facebook data warehouse to 300PB (April 2014). https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb Accessed 30 Dec 2019.
2. Dittrich J, Quiané-Ruiz J-A, Jindal A, Kargin Y, Setty V, Schad J. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). Proc VLDB Endow. 2010;3(1–2):515–29. https://doi.org/10.14778/1920841.1920908.

3. Richter S, Quiané-Ruiz J-A, Schuh S, Dittrich J. Towards zero-overhead static and Adaptive Indexing in Hadoop. VLDB J. 2014;23(3):469–94. https://doi.org/10.1007/s00778-013-0332-z.
4. Schuhknecht FM, Dittrich J, Linden L. Adaptive Adaptive Indexing. In: ICDE; 2018.
5. Bloom BH. Space/time trade-offs in hash coding with allowable errors. Commun ACM. 1970;13(7):422–6. https://doi.org/10.1145/362686.362692.
6. Herodotou H, Babu S. Profiling, what-if analysis, and cost-based optimization of MapReduce programs. PVLDB. 2011;4(11):1111–22.
7. Jahani E, Cafarella MJ, Ré C. Automatic optimization for MapReduce programs. PVLDB. 2011;4(6):385–96.
8. Todorov Marinov M. A bloom filter application for processing big datasets through mapreduce framework. In: 2021 International Conference on Information Technologies (InfoTech), pp. 1–5;2021. https://doi.org/10.1109/InfoTech52438.2021.9548638.
9. Nehme R, Bruno N. Automated partitioning design in Parallel Database Systems. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. SIGMOD '11, pp. 1137–1148. ACM, New York, NY, USA;2011. https://doi.org/10.1145/1989323.1989444.
10. Ameloot TJ, Geck G, Ketsman B, Neven F, Schwentick T. Data partitioning for single-round multi-join evaluation in massively parallel systems. SIGMOD Rec. 2016;45(1):33–40. https://doi.org/10.1145/2949741.2949750.
11. Lu Y, Shanbhag A, Jindal A, Madden S. AdaptDB: Adaptive partitioning for distributed joins. Proc VLDB Endow. 2017;10(5):589–600. https://doi.org/10.14778/3055540.3055551.
12. Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans Comput Syst. 2008;26(2):4–1426. https://doi.org/10.1145/1365815.1365816.
13. Bhushan M, Banerjea S, Yadav SK. Bloom filter based optimization on HBase with MapReduce. In: Data Mining and Intelligent Computing, IEEE, pp. 1–5;2014. https://doi.org/10.1109/ICDMIC.2014.6954230.
14. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Oper Syst Rev. 2010;44(2):35–40.
15. Fan B, Andersen DG, Kaminsky M, Mitzenmacher MD. Cuckoo Filter: Practically better than Bloom. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. CoNEXT '14, pp. 75–88. ACM, New York, NY, USA;2014. https://doi.org/10.1145/2674005.2674994.
16. Frazier J. Why Data Deletion Makes Sense (and Dollars). http://www.ftijournal.com/article/why-data-deletion-makes-sense-and-dollars Accessed 15 Aug 2018.
17. Lee J-G, Han J, Li X, Gonzalez H. TraClass: Trajectory classification using hierarchical region-based and trajectory-based clustering. Proc VLDB Endow. 2008;1(1):1081–94. https://doi.org/10.14778/1453856.1453972.
18. Zamanian E, Binnig C, Salama A. Locality-aware partitioning in parallel database systems. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. SIGMOD '15, pp. 17–30. ACM, New York, NY, USA 2015. https://doi.org/10.1145/2723372.2723718.
19. Hentschel B, Kester MS, Idreos S. Column sketches: A scan accelerator for rapid and robust predicate evaluation. In: Proceedings of the 2018 International Conference on Management of Data. SIGMOD '18, pp. 857–872. ACM, New York, NY, USA;2018. https://doi.org/10.1145/3183713.3196911. http://doi.acm.org/10.1145/3183713.3196911.
20. Sidirourgos L, Kersten M. Column imprints: A secondary index structure. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. SIGMOD '13, pp. 893–904. ACM, New York, NY, USA;2013. https://doi.org/10.1145/2463676.2465306. http://doi.acm.org/10.1145/2463676.2465306.
21. Tian Y, Zou T, Ozcan F, Goncalves R, Pirahesh H. Joins for hybrid warehouses: Exploiting massive parallelism in hadoop and enterprise data warehouses. In: Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015., pp. 373–384;2015. https://doi.org/10.5441/002/edbt.2015.33.
22. Polychroniou O, Sen R, Ross KA. Track join: Distributed joins with minimal network traffic. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14, pp. 1483–1494. ACM, New York, NY, USA;2014. https://doi.org/10.1145/2588555.2610521.
23. Vogels W. Eventually consistent. Commun ACM. 2009;52(1):40–4.
24. He Y, Lee R, Huai Y, Shao Z, Jain N, Zhang X, Xu Z. RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems. In: 2011 IEEE 27th International Conference on Data Engineering, pp. 1199–1208;2011. https://doi.org/10.1109/ICDE.2011.5767933.
25. Chen S. Cheetah: a high performance, custom data warehouse on top of MapReduce. Proc VLDB Endow. 2010;3(1–2):1459–68. https://doi.org/10.14778/1920841.1921020.
26. Tarkoma S, Rothenberg CE, Lagerspetz E. Theory and practice of Bloom filters for distributed systems. IEEE Commun Surv Tutorials. 2012;14(1):131–55. https://doi.org/10.1109/SURV.2011.031611.00024.
27. Liang Y, Yu Y, Ouyang W. Improved big data filtering algorithm based on bloom filter. J Phys. 2020;1629(1):012026. https://doi.org/10.1088/1742-6596/1629/1/012026.
28. Kwon M, Reviriego P, Pontarelli S. A length-aware Cuckoo filter for faster IP lookup. In: 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 1071–1072;2016. https://doi.org/10.1109/INFCOMW.2016.7562258.
29. Cui J, Zhang J, Zhong H, Xu Y. SPACF: a secure privacy-preserving authentication scheme for VANET with Cuckoo filter. IEEE Trans Vehicular Technol. 2017;66(11):10283–95. https://doi.org/10.1109/TVT.2017.2718101.
30. Mahale VV, Pareek NP, Uttarwar VU. Alleviation of DDoS attack using advance technique. In: 2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), pp. 172–176;2017. https://doi.org/10.1109/ICIMIA.2017.7975595.
31. Al-Hisnawi M, Ahmadi M. Deep packet inspection using Cuckoo filter. In: 2017 Annual Conference on New Trends in Information Communications Technology Applications (NTICT), pp. 197–202;2017. https://doi.org/10.1109/NTICT.2017.7976111.

32. Kwon M, Vajpayee S, Vijayaragavan P, Dhuliya A, Marshall J. Use of Cuckoo filters with FD.Io VPP for software IPv6 routing lookup. In: Proceedings of the SIGCOMM Posters and Demos, pp. 127–129. ACM, ???;2017. https://doi.org/10.1145/3123878.3132010.
33. Ren K, Zheng Q, Arulraj J, Gibson G. SlimDB: A space-efficient key-value storage engine for semi-sorted data. Proc VLDB Endow. 2017;10(13):2037–48. https://doi.org/10.14778/3151106.3151108.
34. Xue Q, Chuah MC. Cuckoo-filter based privacy-aware search over encrypted cloud data. In: 2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN), pp. 60–68;2015. https://doi.org/10.1109/MSN.2015.41.
35. Chen H, Liao L, Jin H, Wu J. The Dynamic Cuckoo Filter. In: 2017 IEEE 25th International Conference on Network Protocols (ICNP), pp. 1–10;2017. https://doi.org/10.1109/ICNP.2017.8117563.
36. Sun Y, Hua Y, Jiang S, Li Q, Cao S, Zuo P. SmartCuckoo: A fast and cost-efficient hashing index scheme for cloud storage systems. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference. USENIX ATC '17, pp. 553–565. USENIX Association, Berkeley, CA, USA;2017.
37. Kirsch A, Mitzenmacher M, Wieder U. More robust hashing: Cuckoo hashing with a stash. SIAM J Comput. 2009;39(4):1543–61. https://doi.org/10.1137/080728743.
38. Mitzenmacher M, Pontarelli S, Reviriego P. Adaptive Cuckoo filters. In: Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments, pp. 36–47;2018. https://doi.org/10.1137/1.9781611975055.4.
39. Casares J. Multi-datacenter Replication in Cassandra. (November 2012). https://www.datastax.com/blog/2012/11/multi-datacenter-replication-cassandra Accessed Dec 30;2019.

Publisher's Note