M.Sc. Engg. Thesis

Improving Query Execution Performance for Database of Big Data

by
Sharafat Ibn Mollah Mosharraf

Submitted to

Department of Computer Science and Engineering

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

Bangladesh University of Engineering and Technology (BUET)

Dhaka 1000

September 2018

The thesis titled "Improving Query Execution Performance for Database of Big Data", submitted by Sharafat Ibn Mollah Mosharraf, Roll No. **0412052019**, Session April 2012, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on September 24, 2018.

Board of Examiners

1. _____

Dr. Muhammad Abdullah Adnan
Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

Chairman
(Supervisor)

2. _____

Dr. Md. Mostofa Akbar
Head and Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

Member
(Ex-Officio)

3. _____

Dr. Mohammed Eunus Ali
Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

Member

4. _____

Dr. Rifat Shahriyar
Assistant Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology, Dhaka.

Member

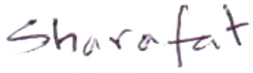
5. _____

Dr. Chowdhury Farhan Ahmed
Professor
Department of Computer Science and Engineering
University of Dhaka, Dhaka.

Member
(External)

Candidate's Declaration

This is hereby declared that the work titled "Improving Query Execution Performance for Database of Big Data" is the outcome of research carried out by me under the supervision of Dr. Muhammad Abdullah Adnan, in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka 1000. It is also declared that this thesis or any part of it has not been submitted elsewhere for the award of any degree or diploma.



Sharafat Ibn Mollah Mosharraf

Candidate

Acknowledgment

Foremost, I am thankful to Almighty Allah for his blessings for the successful completion of my thesis. I would like to express my heartiest gratitude, profound indebtedness and deep respect to my supervisor, Dr. Muhammad Abdullah Adnan, Assistant Professor, Dept. of CSE, BUET, Dhaka, Bangladesh, for his constant supervision, affectionate guidance and great encouragement and motivation. His keen interest on the topic and valuable advices throughout the study was of great help in completing thesis.

I would also want to thank the members of my thesis committee for their valuable suggestions. I thank Dr. Md. Mostofa Akbar, Dr. Mohammed Eunus Ali, Dr. Rifat Shahriyar and specially the external member Dr. Chowdhury Farhan Ahmed.

I am especially grateful to Department of Computer Science and Engineering (CSE) of Bangladesh University of Engineering and Technology (BUET) for providing their support during the thesis work. My sincere thanks goes to CSE Office staffs for providing logistic support to me to successfully complete the thesis work.

Finally, I would like to thank my family: my wife, parents and all of those who supported me for their appreciable assistance, patience and suggestions during the course of my thesis.

Abstract

Performance is a critical concern when reading and writing data from billions of records stored in a Big Data warehouse. Many researchers have proposed improving query execution performance in distributed Big Data systems by introducing efficient techniques such as indexing, caching, filtering, map-reduce, query execution plan, data partitioning, etc. In this thesis, we introduce two other scopes for query performance improvement. One is to improve performance of lookup queries after data deletion in Big Data systems that use the Eventual Consistency model. We propose a scheme to improve performance of lookup queries after data deletion by replacing Bloom Filter with a better probabilistic data structure called Cuckoo Filter that supports deletion of elements. Another scope for query performance improvement is to avoid unnecessary network round-trip for query execution in remote nodes in a Big Data cluster when it is known that the nodes do not have the requested partition of data. We propose a scheme using probabilistic filters that are looked up before delegating a query execution to remote nodes, so that queries resulting in no data can be skipped from passing through the network. We evaluate our schemes with a popular Big Data database (Cassandra) and show that each scheme can improve performance of lookup queries for up to 100%. We also show that the proposed schemes do not degrade performance of other data manipulation queries as a side effect.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This thesis presents two techniques to improve query execution performance in Big Data systems. It critically analyzes existing researches on query performance improvement, proposes schemes to improve performance in cases that have not been addressed till now, and analyzes results of carefully-designed experiments with real data sets.

In this introductory chapter the rationale for this research is explained and an overview of the thesis is provided. The chapter starts off by presenting the motivations and objectives of the research. Then the contributions of the research are briefly outlined. Finally, an overview of the organization of the thesis is provided.

1.1 Motivation

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 [9].

To improve performance, Big Data systems utilize the Eventual Consistency model as well as a probabilistic data structure called Bloom Filter. In recent times, cost minimization and privacy concerns have led Big Data systems to regularly delete data besides reading and writing. Bloom Filter has an important limitation of not supporting deletion of elements from within it. This lim-

itation along with constraints of the Eventual Consistency model prevents the system to delete data from disk immediately upon a deletion query execution. Rather data is actually deleted during a compaction process, which is executed very infrequently due to it being resource-intensive. During the time between data deletion and compaction (typically a few days), lookup queries fetch records from disk that include the deleted rows, then detect and remove the deleted records from the returning result set. This leads to poor query execution performance.

Again, most Big Data systems use Distributed Hash Tables to determine nodes that contain specific data partitions. When the node processing a query requires data from remote nodes, it simply fetches the data from those nodes and then combine and construct the query result. While the existing researches try to improve query performance in various ways, none of those address an important case for improving performance - that when the query executed in remote node does not produce any result at all, if we can know it beforehand, we can avoid a network round trip that in any case would not have returned any result, and consequently query execution time can improve significantly.

In this research, we propose two schemes to improve query execution performance in the aforementioned cases. In one of the schemes, we improve performance of lookup queries after data deletion by replacing Bloom Filter with a better probabilistic data structure called Cuckoo Filter that supports deletion of elements from within it. We evaluate our proposed scheme using a popular Big Data database (Cassandra) that uses both the Eventual Consistency model and Bloom Filter, and show that performance of lookup queries executed after data deletion can improve up to 100% for each deleted record. We also show that our scheme does not degrade performance of other data manipulation queries as a side effect.

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

1.1.1 The Need for Query Optimization in Distributed Big Data Systems

To improve storage and processing efficiency, distributed Big Data systems partition data into several nodes in a cluster. When a client connects to a node and executes a query that require fetching data from multiple nodes and process the data, the node needs to execute that query in other nodes and fetch results. Network latency is a bottleneck in query performance in this case. If the required data does not exist in remote node, the network round-trip becomes an absolute waste. If the node has some efficient method to look up whether the requested data actually exist in remote nodes, before it executes the query in those remote nodes; then unnecessary network latency can be avoided, resulting in significant query performance improvement.

1.1.2 The Need for Deleting Data From Big Data Systems

There are many strong reasons for organizations to delete anywhere from small to large amount of data from their warehouses. In this section we discuss a few most common ones among those reasons.

1.1.2.1 Unnecessary data costs storage (a lot)

Removing unnecessary and duplicate data improves operational efficiency, as duplicate data drive up data volume while slowing processing times and hampering business agility. As an 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 five years [10].

1.1.2.2 Cloud Big Data providers charge for query executions against amount of data processed by those queries

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 [1]. Therefore, the more data

[1] As an example of pricing, Google charges $5 per Terabyte for its BigQuery query execution service (as of Aug. 15, 2018). See https://cloud.google.com/bigquery/pricing.

is there, the costlier each query execution becomes; and hence deleting unnecessary data would cut down costs at a great extent.

1.1.2.3 More data and information attract hackers and other criminals

According to Cisco's Visual Networking Index forecast [11], global information processing traffic will grow at a compound annual growth rate of 20+ percent from 2016 to 2021, resulting in 3.3 ZB of data flow per year by 2021. All these collected data attract hackers and other criminals as personal credit information (which can be either used or sold) becomes more available and accessible.

1.1.2.4 Data analysts waste time working with useless junk data

Not all data that businesses collect are useful. Indeed, as the enterprise's haystack of data climbs ever higher, businesses often do not know what data they possess. Much of the information may be and frequently is junk, and data analysts waste time working with this junk, finding spurious patterns within it, thus hindering the company's decision-making capabilities while incurring needless costs [10].

1.2 Thesis Objectives

Many researchers have proposed improving query execution performance in distributed Big Data systems by introducing efficient techniques such as indexing, caching, filtering, map-reduce, query execution plan, data partitioning, etc. In this research, we introduce two other scopes for improvement.

One of the scopes for improving query performance is avoiding unnecessary network round-trip for query execution in remote nodes in a Big Data cluster when it is known that the nodes do not have the requested partition of data. We propose a scheme using probabilistic filters that are looked up before delegating a query execution to remote nodes so that queries that will result in no data can be skipped from passing through the network. We evaluate our scheme with a popular

Big Data database (Cassandra) and show that performance of lookup queries can improve for up to 100% in cases where data do not exist in remote nodes.

The other scope for improving query performance that is introduced in this research is improving performance of lookup queries after data deletion in Big Data systems utilizing Eventual Consistency model along with Bloom Filters. We replace the Bloom Filter used by the Big Data systems with Cuckoo Filter that allows deletion of entries from within it. When rows are deleted, we delete the corresponding row keys from the Cuckoo Filter. Subsequent queries that try to look up those deleted keys will find that the filter does not contain the keys and hence will be able to avoid costly disk access. We show that the lookup query after data deletion can improve performance by up to 100% using our proposed scheme. We also show that our scheme does not cause performance degradation as a side effect for any lookup or insertion queries whatsoever beyond the case of lookup queries after deletion.

1.3 Contributions

The contributions of this research are as follows:

1. We propose a 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.

2. We propose another scheme to improve query execution performance 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.

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 schemes and show that the schemes do not degrade query performance in other cases as a side effect.

1.4 Thesis Organization

The organization of the rest of the thesis is as follows.

In **Chapter 2**, we have presented relevant background studies on various techniques of query performance improvement in Big Data systems, along with detailed analysis of Bloom Filter and Cuckoo Filter - probabilistic data structures that are at the core of our proposed schemes. We also discuss about Cassandra, our experimental Big Data system, and how it utilizes Bloom Filter to improve query performance.

In **Chapter 3**, we have presented literature review on research works related to our study.

Chapter 4 explains our first proposed scheme in detail, along with experimental results and their analyses. It also discusses the challenges in implementing our proposed scheme and how we addressed those challenges.

Chapter 5 explains our second proposed scheme in detail, along with experimental results and their analyses. Similar to the previous chapter, it also discusses the challenges in implementing our proposed scheme and our solutions to those challenges.

Finally, **Chapter 6** concludes our thesis. This chapter also includes the outlines of some future works related to this dissertation.

Chapter 2

Background

In this chapter, we present background studies related to our research. In Section 2.1 we state the primary bottlenecks of query performance. To improve it, we present available techniques in Section 2.2. We focus on utilizing *Probabilistic Data Structures* to improve query performance and discuss two such data structures in Sections 2.3 and 2.4. Then we briefly describe Big Data in Section 2.5. Finally, in Section 2.6 we state why we have chosen Cassandra as our experimental Big Data database and briefly describe how it utilizes Bloom filters.

2.1 Query Performance

Big Data deals with millions/billions of records, and reading/writing millions of records is slow. It is a well-known fact that disk access overhead is the primary bottleneck of querying in a database, around 99% of the total query processing time is spent during disk IO. As an example, Figure 2.1 demonstrates the time duration of several query execution processing components, using a sample query executed during our experimentation. From the figure it can be seen that while all the components take time in milliseconds, the disk scanning component takes time in seconds, demonstrating the huge overhead of disk access costs.

Section 2.2 briefly describes available techniques that can be used to minimize disk access overhead.

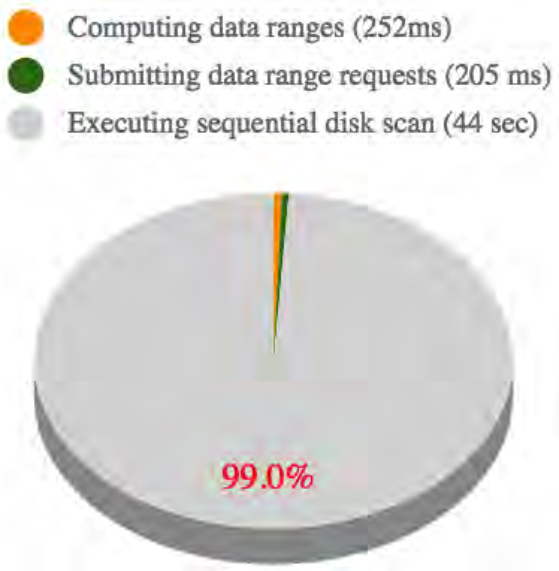


Figure 2.1: Performance of query components. All the components take milliseconds to complete while the disk scanning component takes seconds, which is 99% of the total query execution time. The components and time duration have been collected by running a query from one of our experiments.

2.2 Minimizing Disk Access Overhead

Many techniques have been invented to improve disk access performance or reduce disk access overhead. Some are hardware improvements while others are software-based. In this section, we briefly describe the available techniques along with their pros and cons.

2.2.1 Using Solid-State Drives

Solid-state drives store data persistently using integrated circuits as opposed to magnetic memory used in the traditional hard disk drives. Solid-state drives have been shown to improve disk access performance by two-fold (Figure 2.2) [1]. However, it increases the cost about 2.5 times [1]. Note that access time is still in *seconds* instead of *milliseconds*, which still keeps disk scan responsible for 99% of total query performance.

[1] As an example, Amazon AWS charges $0.125 per GB-month for SSDs, whereas it charges $0.045 per GB-month for HDDs (as of Aug. 15, 2018). See https://aws.amazon.com/ebs/pricing.

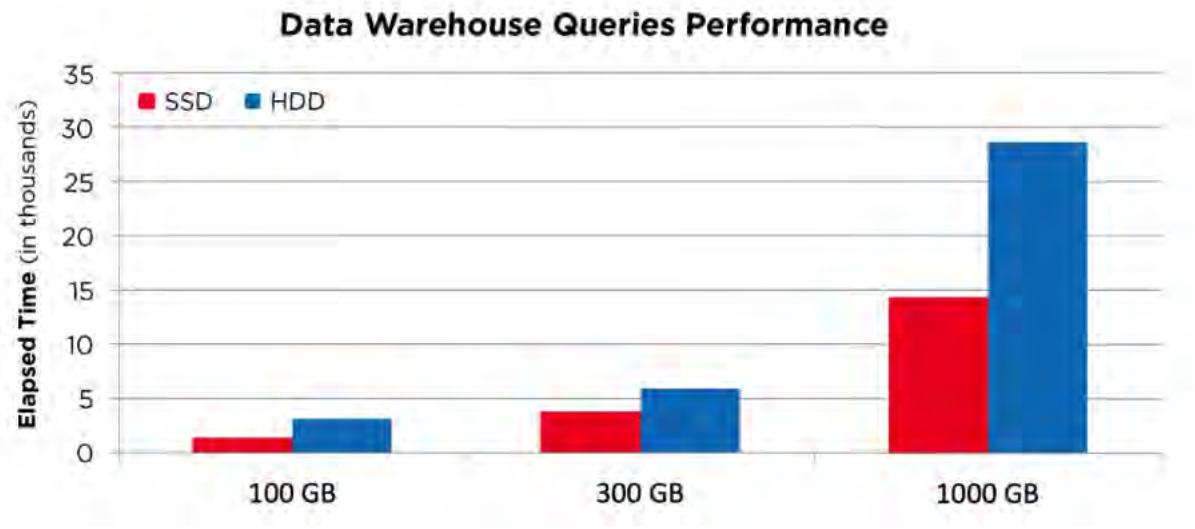


Figure 2.2: Data warehouse query performance on SSD vs. HDD. SSD performs $2x$ better than HDD [1].

2.2.2 Using More RAM

Caching data in RAM will speed up query performance a lot. Doing a sequential access in RAM performs 3-5 times better than SSD and 10-15 times better than HDD; whereas doing a random access in RAM performs 1,000 times better than SSD and 100,000 times better than HDD; although RAM is 10 times costlier than an SSD and 100 times costlier than an HDD (see Figure 2.3) [12]. Hence, putting all the data of a Big Data database into RAM is obviously cost-prohibitive.

2.2.3 Using Indexes

A *Database Index* is a data structure that improves data retrieval performance, but at the cost of extra writes and storage space to maintain the data structure. Indexes are pretty common in Database Management Systems. It does improve data retrieval performance for up to 2,000 times (see Figure 2.4) [2], but then data insertion, update and deletion performance can get decreased down to 1/1,000 times (see Figure 2.5) [3]. Also, the storage space an index takes is not little (the tiniest index for a single column can take at least 1.3 GB of memory for 100M rows in a

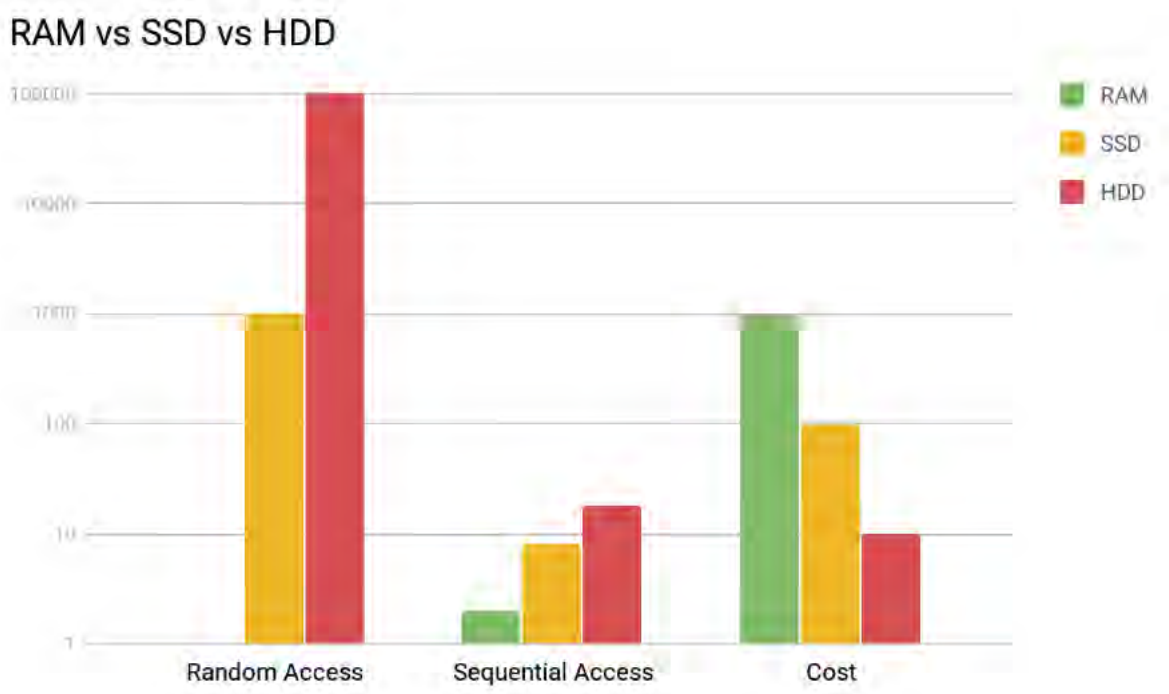


Figure 2.3: Illustrative chart based on arbitrary relative values to visualize the differences among RAM, SSD and HDD.

table [2]), and hence loading all indexes into RAM for faster access is cost-prohibitive. Therefore, looking up indexes from disk storage takes us back to the initial issue of disk access being the major bottleneck of query performance.

2.2.4 Using Probabilistic Data Structures - Filters

Probabilistic data structures cannot provide an exact answer, instead they provide a reasonable approximation of the answer and a way to estimate the approximation. They are extremely useful for Big Data and streaming applications (such as network applications) as they dramatically decrease the amount of storage needed in comparison to data structures that provide exact answers.

A *Probabilistic Filter* provides approximate answer to membership queries. Querying a set for a set membership results in either *true negative* (i.e. reporting an inserted element to be ab-

[2] Calculating index size using the simplified formula: $IndexSize = rows \times (header + indexedColumns + primaryKeyColumns)$, assuming the values $rows = 100M$, $header = 6bytes$, $indexedColumns = 1$ (4-byte integer), $primaryKeyColumns = 1$ (4-byte integer).

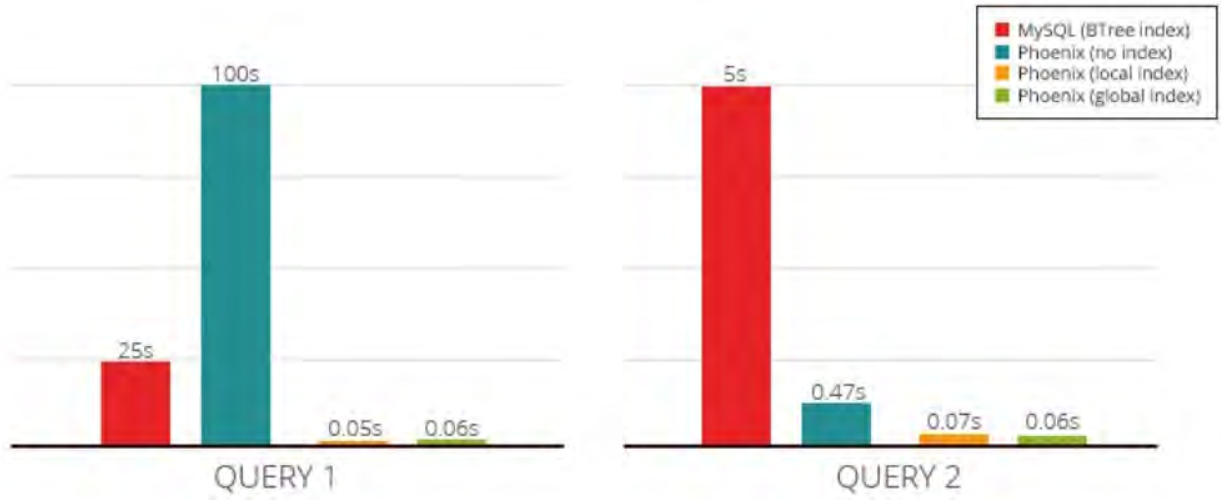


Figure 2.4: Data read performance improvement using index. Index can improve data retrieval performance for up to $2000x$ [2].

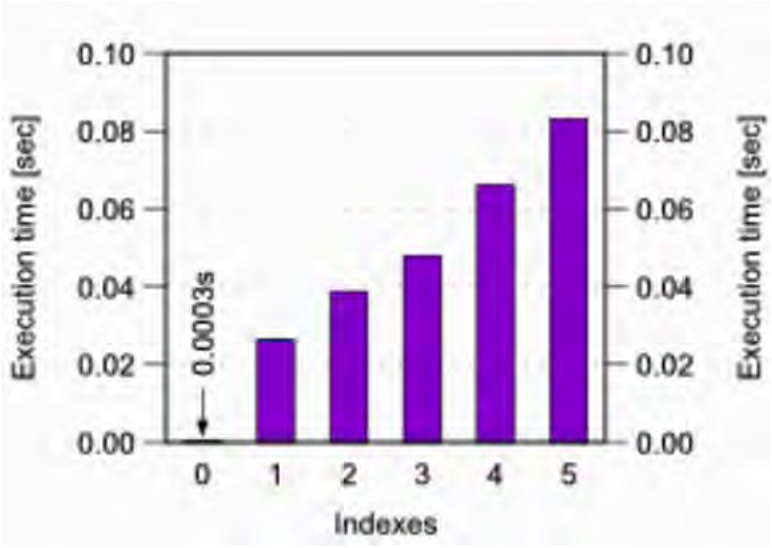


Figure 2.5: Data write performance decreases due to using index. Data insertion, update and deletion performance can get decreased down to 1/1,000 times due to using indexes [3].

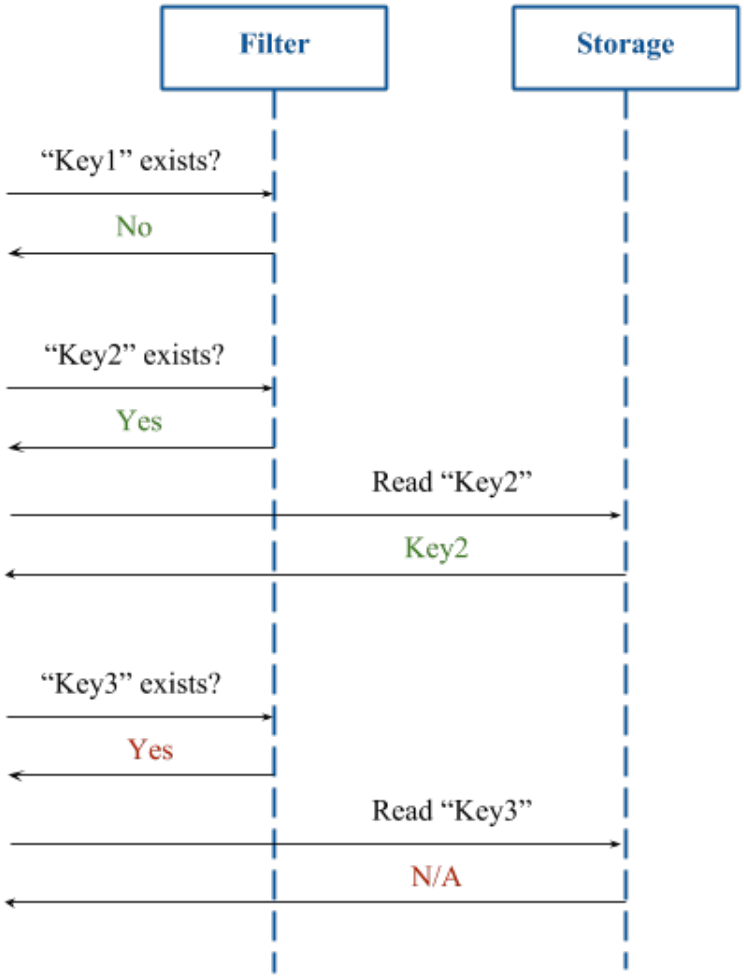


Figure 2.6: How a *Probabilistic Filter* works.

sent from the set) or *false positive* (i.e. claiming an element to be part of the set when it was not inserted). The filter can be queried before accessing disk to know if the disk contains records matching the query. Thus, a lot of unnecessary disk access can be prevented if a probabilistic filter is used. Figure 2.6 shows how a probabilistic filter works and how it can prevent unnecessary disk access.

In Section 2.3, we describe the most popular probabilistic filter out there - the Bloom Filter. In Section 2.4 we describe Cuckoo Filter - a better filter in terms of performance and storage efficiency.

2.3 Bloom Filter

The technique of probabilistic filters was first published by B. H. Bloom in 1970 [13], which since has become known as Bloom Filter. This is the most popular probabilistic filter data structure used till now because of its design simplicity, performance and extremely small memory footprint. In Section 2.3.1 we describe how Bloom Filter works, followed by Section 2.3.2 where we discuss the usefulness and limitations of Bloom Filter. Finally, in Section 2.3.3 we state a few examples of how Bloom Filter is used in Big Data.

2.3.1 How Bloom Filter Works

A Bloom Filter is a simple bit-array data structure with a number of hash functions that determine which data should set the bit of which index of the array.

In Figure 2.7 we show step by step on how insertion in a Bloom Filter works. Let us suppose our example Bloom Filter has 10 buckets (i.e. a 10-bit array) [Figure 2.7 (a)] along with two hash functions - FNV [14] and Murmur [15] to determine data index in the filter.

If we insert the key *amazing* into the filter, we get $FNV(amazing) = 4$ and $Murmur(amazing) = 2$; so we set the indices 2 and 4 of the filter [Figure 2.7 (b)].

Similarly, let us insert *wonderful* and *filter*. For *wonderful*, $FNV(wonderful) = 5$ and $Murmur(wonderful) = 0$, so we set the bits 0 and 5 [Figure 2.7 (c)]. For *filter*, $FNV(filter) = 5$ and $Murmur(filter) = 6$, so we set the bits 5 (which is already set) and 6 [Figure 2.7 (d)].

After inserting *amazing*, *wonderful* and *filter*, the state of the filter is that the indices 0, 2, 4, 5, 6 are set [Figure 2.8 (a)].

Now, let us inquire for memberships and see how Bloom Filter works. When we inquire for *amazing*, the filter calculates $FNV(amazing) = 4$ and $Murmur(amazing) = 2$, and checks if any of the bits are set. As both the bits are set, the filter responds with *maybe*, that is, the key *amazing* may be in the set, but it is not absolutely sure about that [Figure 2.8 (b)].

Now let's inquire for a key that hasn't been really inserted - *bad*. The filter calculates $FNV(bad) = 6$ and $Murmur(bad) = 3$, and finds that none of the indices are set; so it confi-

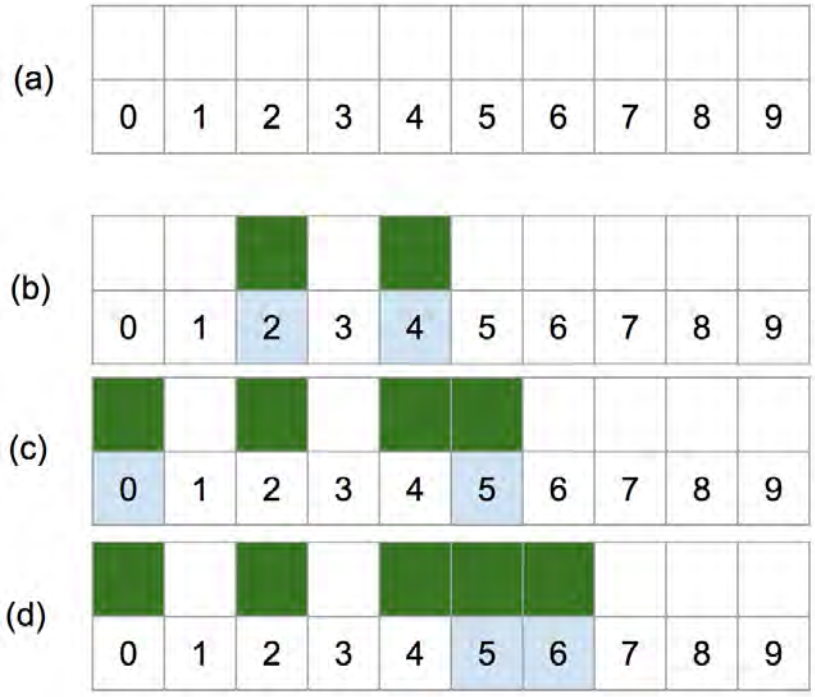


Figure 2.7: How data insertion in Bloom Filter works.

dently responds with *negative*, that is, the key *bad* absolutely isn't present in the set [Figure 2.8 (c)].

As a final example, querying for *true*, which is really not in the set, the filter calculates $FNV(true) = 2$ and $Murmur(true) = 0$, and finds that the index 0 is set; so it responds with *maybe*, that is, the key *true* may be in the set, but it is not absolutely sure about that [Figure 2.8 (c)]. This is where it shows the characteristic of *false positive* responses.

2.3.2 Pros and Cons of Bloom Filter

Bloom Filter has the following advantages:

1. Extremely low memory footprint. For example, it requires only 2.26MB at 1% FPP (false-positive probability) for 1M keys using 2 hash functions, regardless of the size of the entries [3].

2. Appropriately sizing the bloom filter relative to the number of keys stored, false positive

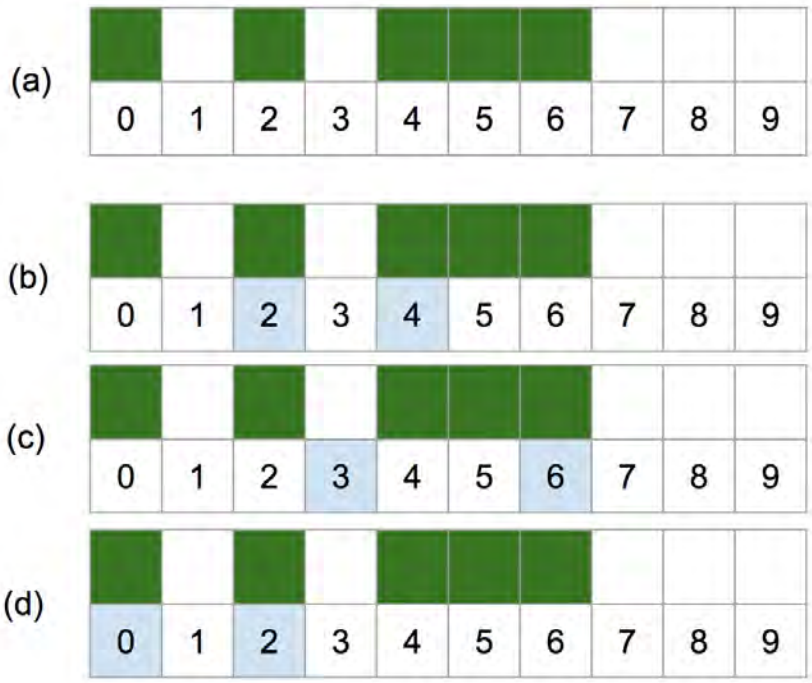


Figure 2.8: How data querying in Bloom Filter works.

rate can be kept reasonably low. For example, it results in 0.1% FPP for 1M keys and 2 hash functions, with 7.42MB total memory footprint [3].

Bloom Filter has the following limitations:

1. Unable to delete items from the filter.
2. Unable to count number of insertions of a single item.
3. The more load, the more false-positive rate.
4. As it cannot definitively conclude whether a key is stored in it, it cannot be automatically resized without creating a brand new Bloom Filter and re-inserting all of the keys through a source of truth.

[3] Calculated using the formula $FPP = (1 - e^{-\frac{k(n+0.5)}{m-1}})^k$, where k = Number of hash functions, n = Number of elements, m = Total number of bits in filter. [16]

2.3.3 Bloom Filter in Big Data

Bloom Filter is being used in Big Data systems in various ways to improve data lookup performance. The most extensive use of Bloom Filters in Big Data is to minimize disk lookups [17]. Popular Big Data systems like Google BigTable [18] and Apache Cassandra [19] use Bloom Filters for this purpose.

Bloom Filter has also been shown to improve performance of join operations in Big Data [20], which is usually performed using the MapReduce technique. Bloom Filter has been integrated into the MapReduce system of HBase [21] and Hadoop [22] and shown that the performance dramatically increases.

2.4 Cuckoo Filter

Bin Fan and others proposed Cuckoo Filter [8] and showed it to be a better filter than Bloom in terms of performance, false-positive rate and storage space per key. In Section 2.4.1 we describe how Cuckoo Filter works, followed by Section 2.4.2 where we discuss the advantages and limitations of Cuckoo Filter.

2.4.1 How Cuckoo Filter Works

The Cuckoo Filter utilizes a Cuckoo Hash [23] as its underlying data structure. Unlike a Bloom Filter that simply stores a bit against each key, Cuckoo Filter stores the *fingerprint* of a key.

The basic operations of a Cuckoo Filter is that for a key x, it calculates fingerprint $f = fingerprint(x)$, bucket index $i_1 = hash(x)$, alternate bucket index $i_2 = i1 \oplus hash(f)$. During lookup, if i_1 or i_2 contains f, then the key may exist in the filter (which can be a false positive if two different keys results in the same fingerprint), otherwise the key definitely does not exist in the filter. During insertion, if i_1 is empty, it is populated with f, if not, then i_2 is populated with f if that is empty. In case neither i_1 nor i_2 is empty, then the Cuckoo Hashing algorithm as shown in Algorithm 1 is executed to evict an existing from the indices and putting that into its alternate index location.

Algorithm 1: Cuckoo Filter insertion algorithm (`Insert(`x`)`) [8].

```
f = fingerprint(x);
i1 = hash(x);
i2 = i1 ⊕ hash(f);
if bucket[i1] or bucket[i2] has an empty entry then
    add f to that bucket;
    return Done;
end
// must relocate existing items;
i = randomly pick i1 or i2;
for n = 0; n < MaxNumKicks; n++ do
    randomly select an entry e from bucket[i];
    swap f and the fingerprint stored in entry e;
    i = i ⊕ hash(f);
    if bucket[i] has an empty entry then
        add f to bucket[i];
        return Done;
    end
end
// Hashtable is considered full;
return Failure;
```

In Figure 2.9 we show step by step on how insertion in a Cuckoo Filter works. Let's suppose our example Cuckoo Filter has 10 buckets (i.e. a 10-bit array) [Figure 2.9 (a)].

While inserting the key *amazing* into the filter, suppose we get $fingerprint(amazing) = a$ and $hash(amazing) = 0$; as the index 0 is empty, so we set $bucket[0] = a$ in the filter [Figure 2.9 (b)].

Let us now insert *wonderful* and suppose we get $fingerprint(wonderful) = b$, $hash(wonderful) = 0$ and $bucket_alt(wonderful) = 3$; we can see that the index 0 is not empty but the index 3 is. So we set $bucket[3] = a$ in the filter [Figure 2.9 (c)].

If we now insert *filter* and we get $fingerprint(filter) = x$, $hash(filter) = 0$ and $bucket_alt(filter) = 3$, we can see that none of the indices 0 and 3 are empty. In that case, suppose we want to evict the fingerprint in the index 3, which is b. We calculate $bucket_alt(b) = 7$, so we set $bucket[7] = b$ and $bucket[3] = x$ in the filter [Figure 2.9 (d)].

After inserting *amazing*, *wonderful* and *filter*, the state of the filter is that the indices 0, 3 and

Figure 2.9: How data insertion in Cuckoo Filter works.

7 contains the fingerprints a, b and x respectively [Figure 2.10 (a)].

Now, let us inquire for memberships and see how Cuckoo Filter works. When we inquire for *amazing*, the filter calculates $fingerprint(amazing) = a$ and $hash(amazing) = 0$, and checks if $bucket[0] = a$ [Figure 2.10 (b)]. As that is the case here, the filter responds with *maybe*, that is, the key *amazing* may be in the set, but it is not absolutely sure about that.

Now let us inquire for a key that hasn't been really inserted - *bad*. Suppose the filter calculates $fingerprint(bad) = z$, $hash(bad) = 3$ and $bucket_alt(bad) = 6$; and finds that none of the indices contain z [Figure 2.10 (c)]; so it confidently responds with *negative*, that is, the key *bad* absolutely is not present in the set.

As a final example, querying for *true*, which is really not in the set, suppose the filter calculates $fingerprint(true) = a$, $hash(true) = 0$ and $bucket_alt(true) = 3$; and finds that $bucket[0] = a$ [Figure 2.10 (d)]; so it responds with *maybe*, that is, the key *true* may be in the set, but it is not absolutely sure about that. This is where it shows the characteristic of *false positive* responses.

Figure 2.10: How data querying in Cuckoo Filter works.

2.4.2 Pros and Cons of Cuckoo Filter

Cuckoo Filter has the following advantages as stated in [8]:

1. Supports Delete and Count operations.
2. No false-negatives during Delete operations.
3. Fewer bits per entry when false positive rate is 3%
4. Cuckoo Filter can maintain a stable false positive rate with load up to around 95%

Cuckoo Filter has an important limitation though. Due to the algorithmic complexity of the insertion algorithm, as more items need to be kicked and placed into alternate buckets, the insertion time increases. And if exhausted, the filter needs to be resized.

However, the limitation can be mitigated by either using a large filter size as per estimation of the number of items to be inserted into the filter, or using a variation of Cuckoo Hashing called *SmartCuckoo* [24], which efficiently predetermines insertion failures without paying a high cost

Table 2.1: Comparison between Cuckoo Filter and Bloom Filter [6]. Cuckoo Filter outperforms Bloom Filter in almost all aspects.

Aspect	**Cuckoo Filter**	**Bloom Filter**
Insert	Variable. $O(1)$ amortized. Longer as load factor approaches capacity.	Fixed. $O(k)$.
Lookup	$O(1)$	$O(k)$
Count	$O(1)$	*Unsupported*
Delete	$O(1)$	*Unsupported*
Bits per entry	Smaller when desired false positive rate $\leq 3\%$.	Smaller when desired false positive rate $> 3\%$.
Availability	Moderately available (as of mid 2018).	Widely available.

of carrying out step-by-step probing. In case of Cassandra, it does not have this problem as it can correctly estimate the number of items to be inserted and hence can instantiate Cuckoo Filters with appropriate sizes.

2.4.3 Comparison Between Cuckoo Filter and Bloom Filter

Table 2.1 summarizes the differences between Cuckoo Filter and Bloom Filter and shows that Cuckoo Filter outperforms Bloom Filter in almost all aspects.

2.5 Big Data

Big Data is the Information asset characterised by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value [25].

Big Data has the following characteristics [25]:

1. **Volume:** The quantity of generated and stored data
2. **Variety:** The type and nature of the data - text, images, audio, video etc.

3. **Velocity:** Big Data is often available in real-time

4. **Veracity:** The data quality of captured data can vary greatly, affecting the accurate analysis

5. **Variability:** Inconsistency of the data set can hamper processes to handle and manage it

NoSQL databases are considered to be Big Data databases due to possessing the aforementioned characteristics.

2.6 Cassandra

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 [26]. Cassandra offers robust support for clusters spanning multiple datacenters [27] with asynchronous masterless replication allowing low latency operations for all clients [26].

Cassandra is open-source [28] and is the most popular Column-Store Big Data database as of July 2018 according to DB-Engines ranking [29].

The source code base of Cassandra is organized and clean. Also, querying the database is very simple compared to other Big Data databases, along with having performance tracing feature built-in. These characteristics along with the aforementioned facts were the driving factors for considering Cassandra as our experimental Big Data database.

In Sections 2.6.1 and 2.6.2 we briefly discuss how Cassandra utilizes Bloom Filter during data write-read to-from disk storage. Finally, in Section 2.6.3 we also discuss about the distributed system architecture Cassandra uses to provide data to clients in response to a query.

2.6.1 How Cassandra Uses Bloom Filter While Writing Data

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending in with a write of data to disk (Figure 2.11):

- Logging data in the commit log

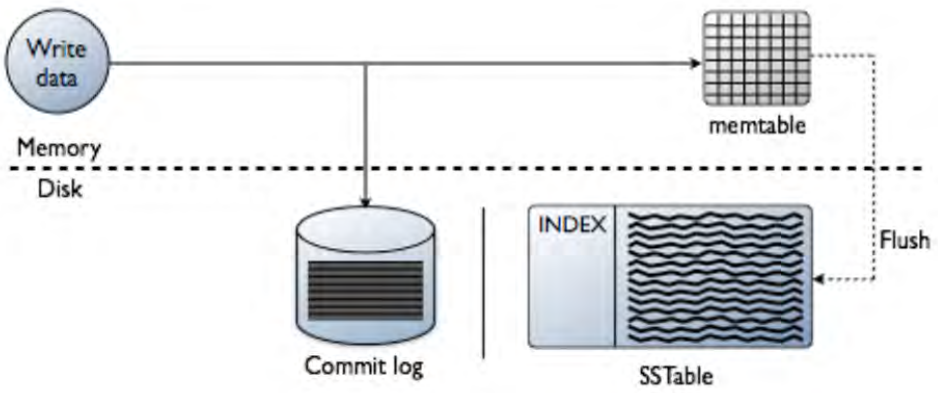


Figure 2.11: Cassandra write path [4].

- Writing data to the memtable
- Flushing data from the memtable
- Storing data on disk in SSTables

As it is possible for Cassandra to store data of a single partition into multiple SSTables, for reading efficiency, it prepares a Bloom Filter for each SSTable when an SSTable is flushed into disk. The details of how this Bloom Filter is utilized to improve lookup performance is discussed in the next Section 2.6.2.

2.6.2 How Cassandra Utilizes Bloom Filter While Reading Data

Cassandra processes data at several stages on the read path to discover where the data is stored, starting with the data in the memtable and finishing with SSTables (Figure 2.12):

- Check the memtable.
- Check row cache, if enabled.
- Checks Bloom filter.
- Checks partition key cache, if enabled.

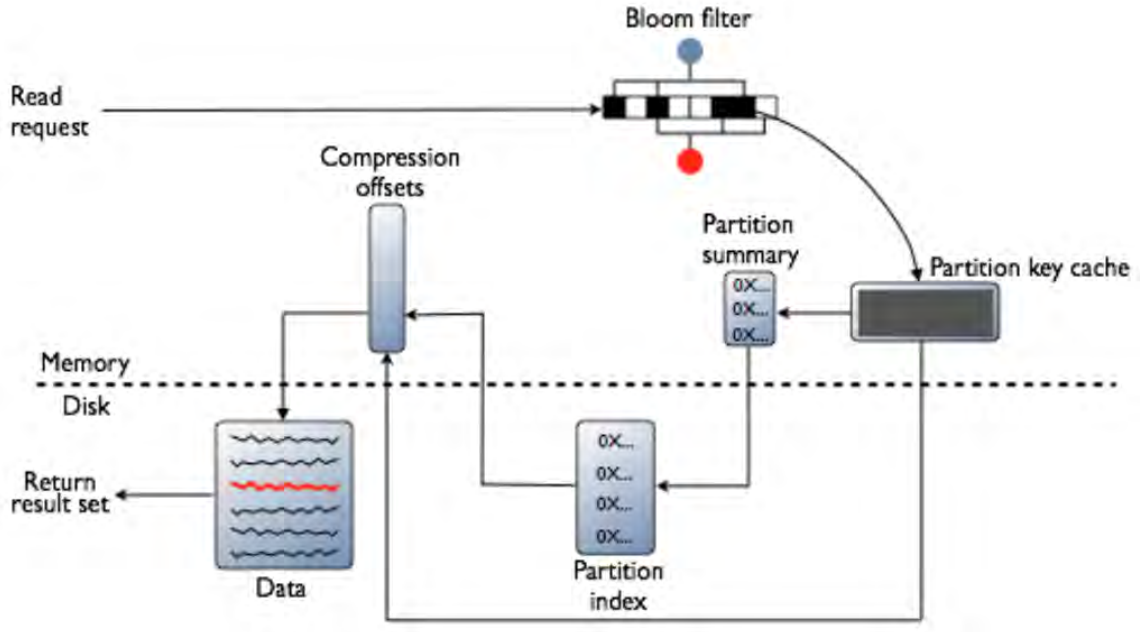


Figure 2.12: Cassandra read path [5].

- Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not. If the partition summary is checked, then the partition index is accessed.
- Locates the data on disk using the compression offset map.
- Fetches the data from the SSTable on disk.

Cassandra checks the Bloom filter to discover which SSTables are likely to have the request partition data. It speeds up the process of partition key lookup by narrowing the pool of keys. However, because the Bloom filter is a probabilistic function, it can result in false positives. Not all SSTables identified by the Bloom filter will have data. If the Bloom filter does not rule out an SSTable, Cassandra proceeds to retrieve the data from the SSTable.

Table 2.2: Policies determining which Cassandra node in a cluster a client should connect to [7].

Policy Name	**Description**
RoundRobinPolicy	Client connects to different nodes for each query in a round-robin fashion.
DCAwareRoundRobinPolicy	Client queries nodes of the local data-center in a round-robin fashion; optionally, it can also try a configurable number of hosts in remote data centers if all local hosts failed.
TokenAwarePolicy	Client requests will be routed in priority to the local replicas that own the data that is being queried.
LatencyAwarePolicy	Collects the latencies of queries to each host, and will exclude the worst-performing hosts from query plans.

2.6.3 Background: How Cassandra Cluster Returns Data Against Client Application Queries

A Cassandra cluster can have multiple nodes, each of which divides partition ranges among them to store. Which particular node a client application will connect to depends on the configuration of the Cassandra driver. The Cassandra driver provides a number of policies to choose specific node to connect to, as described in Table 2.2.

Any policy other than the TokenAwarePolicy is susceptible to potential latency that can occur due to the node a client is connected to not owning the partition of data the client requests for. In this situation, the node fetches the data from the node that owns the partition and relays it to the client. Figure 2.13 and Figure 2.14 shows the difference between the query-result path when the node a client is connected to owns and does not own, respectively, the partition of data the client requested for.

The default policy is a combination of *TokenAwarePolicy* and *DCAwareRoundRobinPolicy*. However, it has two important limitations as discussed in Section 2.6.4.

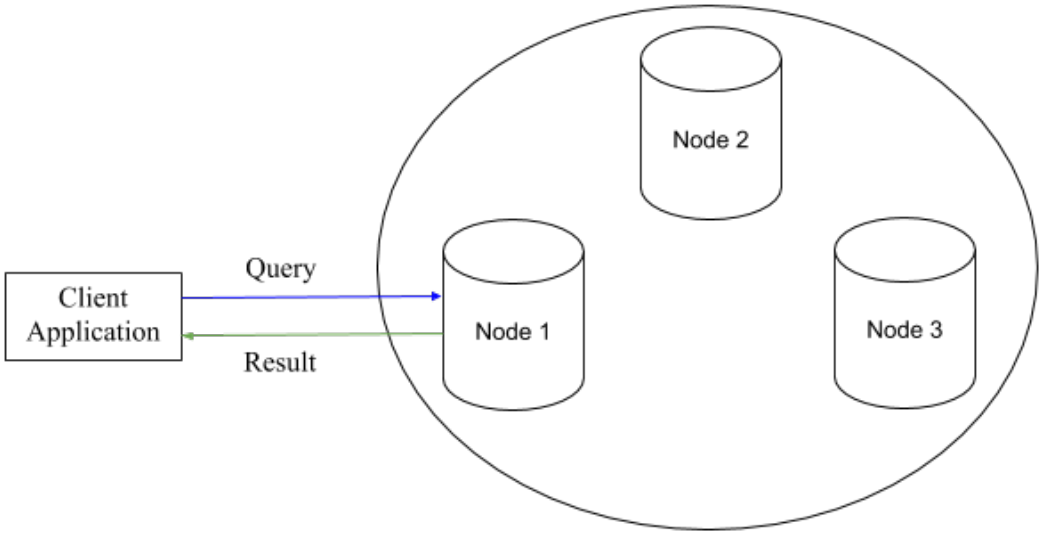


Figure 2.13: Query execution path when Cassandra node *owns* partition of data requested by client.

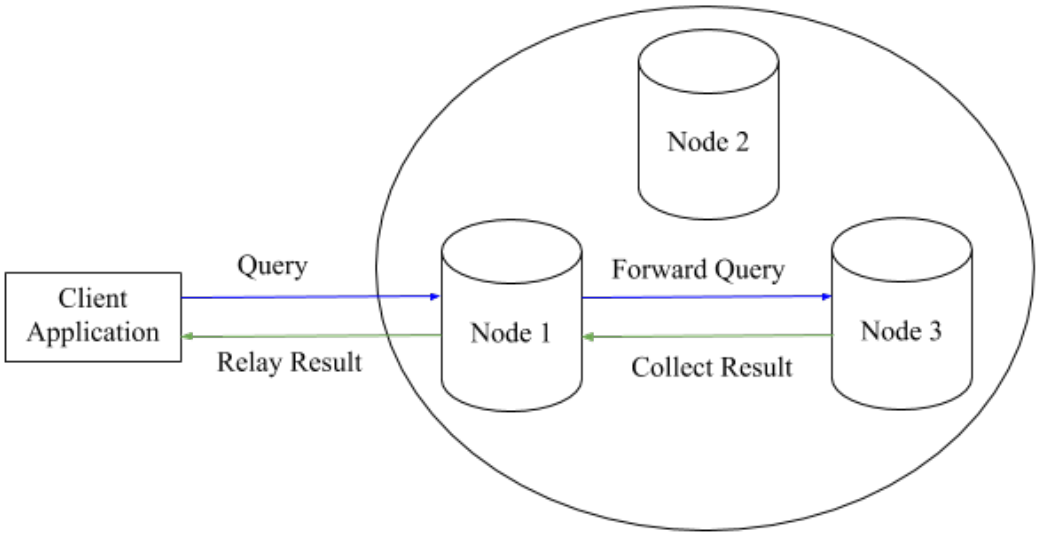


Figure 2.14: Query execution path when Cassandra node *does not own* partition of data requested by client.

2.6.4 Limitations of TokenAwarePolicy

1. It requires that the client executes a query in such a way as the driver is able to determine the KeySpace and RoutingKey (filter columns) from the query. There are multiple ways of executing a query and the client has the burden to make sure the token is able to determine the KeySpace and RoutingKey correctly for each way. [3]

2. In case of `WHERE...IN` queries, the node may not own some partitions of the keys specified in the query and hence in any case needs to collect the required data from the correct nodes.

2.7 Summary

In this chapter, we have discussed about various query optimization techniques, Bloom Filter and Cuckoo Filter as probabilistic data structures and how a popular Big Data system (Cassandra) utilizes Bloom Filters. The next chapter will briefly discuss on contemporary research works on query performance optimization in Big Data.

Chapter 3

Literature Review

Improving query performance in Big Data systems is challenging. This chapter provides an overview of the research works related to our topic - improving query performance in Big Data Systems.

3.1 Researches Related to Improving Query Performance in Big Data Systems

There are two primary factors that affect query performance in a distributed Big Data database 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 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.

3.1.1 Researches Related to Improving Query Performance Within a Single Node

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

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 [30] [31] [32].

Probabilistic filters have been used to improve query execution performance by avoiding unnecessary disk storage access. Google BigTable [18], Apache HBase [21] and Apache Cassandra [19] are the popular Big Data solutions that utilize Bloom Filter, the most popular probabilistic filter.

3.1.2 Researches Related to Improving Query Performance Within a Cluster of Nodes

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 [33] [34] [35], other researches propose schemes to improve query performance for specific categories of data (e.g. geo-spatial [36], locality-aware [37] etc.).

3.1.3 Researches Related to Improving Query Performance by using Efficient Query Execution Plans

Many researches have been carried out that provide query execution plan models that can improve query performance through efficient MapReduce operations. 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, a few notable ones being [38] [39] [40].

3.2 Areas of Query Performance Optimization in Our Research

In our research, we propose improving query performance in two cases not addressed by the existing researches. In sections 3.2.1 and 3.2.2 we discuss those cases. Afterwards, in Sections 3.3 and 3.5 we discuss research works related to our proposed methods.

3.2.1 Improving Query Performance in Big Data Systems That use Eventual Consistency Model

Big Data systems that offer high-performance query execution service usually utilize Bloom Filters along with *Eventual Consistency* model for performance improvement. 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*. 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.

We propose a scheme to improve performance of lookup queries after data deletion by replacing Bloom Filter with Cuckoo Filter that allows deletion of entries from within it. When rows are deleted, we delete the corresponding row keys from the Cuckoo Filter. Subsequent queries that try to look up those deleted keys will find that the filter does not contain the keys and hence will be able to avoid costly disk access. We show that the lookup query after data deletion can improve performance by up to 100% for each deleted record using our proposed scheme.

3.2.2 Improving Query Performance in Big Data Systems by Introducing Node Filter

Most Big Data systems use Distributed Hash Tables to determine nodes that contain specific data partitions. When the node processing a query requires data from remote nodes, it simply fetches the data from those nodes and then combine and construct the query result. While the existing researches try to improve query performance in various ways, none of those address an important case for improving performance - that when the query executed in remote node does not produce any result at all, if we can know it beforehand, we can avoid a network round trip that in any case would not have returned any result, and consequently query execution time can improve significantly. We propose a scheme that can detect absence of data against a partition key from within the node processing the query, thus avoid a network trip cost that would otherwise fetch no result for the query. We show that based on the percentage of partition key that results in no data from different nodes, our scheme can improve query execution performance 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 inserted element to be absent from the set) or *false positive* (i.e. claiming an element to be part of the set when it was not inserted). 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.

3.3 Bloom Filter and Its Major Variants

3.3.1 Bloom Filter

The technique of filters was first published by B. H. Bloom [13], which became known as Bloom Filter. Though it has an extremely small memory footprint, Bloom Filter has the disadvantages of not supporting count and deletion operations on items in the filter, and ascension of false positive rate with increased load. Several variations of Bloom Filter have been proposed to address these issues (see [17] for a comprehensive list), but none can address all the issues of Bloom Filter at

the same time. In this section we describe in short some of the major variations of Bloom Filter and how they fail to fully address all the limitations of Bloom Filter.

3.3.2 Counting Bloom Filter

Counting Bloom Filter [41] brings support for count and deletion operations. Rather than storing a single bit per bucket, it stores count of items placed in that bucket.

However, it takes about four times more memory than a regular Bloom Filter, and in some scenarios, it can introduce false-negatives during delete operation [42].

3.3.3 *d*-Left Counting Bloom Filter

d-Left Counting Bloom Filter [43] is functionally equivalent to Counting Bloom Filter. It uses the d-left hashing scheme [44]. It takes approximately half the memory space of a counting bloom filter. Scalability issue does not occur in this data structure, too. Because once the designed capacity is exceeded, the keys could be reinserted in a new hash table of double size.

However, it takes twice more memory than a regular Bloom Filter. And it also fails to address the false-negative problem of Counting Bloom Filter.

3.3.4 Deletable Bloom Filter

Deletable Bloom Filter [45] is also functionally equivalent to Counting Bloom Filter. It does solve the false-negative issue, but at the expense of false-positive rate and element deletability. Element deletability and false positive rate depend on how much memory space one is willing to invest.

3.3.5 Spectral Bloom Filter

Spectral Bloom Filter [46] supports approximate frequency count of keys. Its memory space slightly larger than regular Bloom Filter. The error rate of frequency count is small. However,

the more error rate is reduced, the less deletion can be performed. And the false-negative issue is still there with it.

3.4 Bloom Filter in Big Data

Bloom Filter is being used in Big Data systems in various ways to improve data lookup performance. The most extensive use of Bloom Filters in Big Data is to minimize disk lookups [17]. Popular Big Data systems like Google BigTable [18] and Apache Cassandra [19] use Bloom Filters for this purpose.

Bloom Filter has also been shown to improve performance of join operations in Big Data [20], which is usually performed using the *MapReduce* technique. Bloom Filter has been integrated into the MapReduce system of HBase and Hadoop [21] and shown that the performance dramatically improves.

3.5 Cuckoo Filter

More recently, Bin Fan and others proposed Cuckoo Filter [8] that supports count and deletion operations, does not introduce false negatives during deletion operation, has fewer bits per entry for optimum false positive rate, and can maintain stable false positive rate with higher load up to 95%.

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. Although the limitation can be mitigated by using a large filter size as per estimation of the number of items to be inserted into the filter, researches have been carried out to devise variations of Cuckoo Filter or Cuckoo Hash that can be used to reduce the probability of insertion failure in more efficient ways. *SmartCuckoo* [24] efficiently predetermines insertion failures without paying a high cost of carrying out step-by-step probing. A. Kirsch et al. [47] showed that the failure probability can be dramatically reduced by the addition of a very small constant-sized *stash*.

There exist other variations that improve upon Cuckoo Filter. *Adaptive Cuckoo Filter* [48] is able to significantly reduce the false positive rate by reacting to false positives, removing them for future queries. Resizing a Cuckoo Filter dynamically with efficient memory and space utilization can be done using *Dynamic Cuckoo Filter* [49] or Dynamically Adjustable Capacity Cuckoo Filter (DACF) [50]. For storage-critical cases, *D-Ary Cuckoo Filter* [51] can save up to one bit per element, though at the cost of increased lookup and insertion performance. Position-aware Cuckoo Filter [52] improves false positive rate of the filter, as does the Length-Aware Cuckoo Filter [53] [54]. 2-3 Cuckoo Filter [55] can answer set-intersection queries fast, as well as list triangles in a graph very fast.

3.6 Cuckoo Filter in Networking and Security

Cuckoo Filter has been utilized in many researches in the area of networking and security. In this section we briefly describe some of the major contributions in this area that used Cuckoo Filter.

3.6.1 Secure Privacy-Preserving Authentication Scheme

Message authentication is a key concern in Vehicular ad-hoc networks (VANETs). Messages should be signed and verified before they could be trusted and passed among entities. Existing solutions either rely heavily on a tamper-proof hardware device or cannot satisfy the security requirement. This paper [56] proposes a scheme that is based on software without relying on any special hardware. It uses Cuckoo filter and binary search methods to achieve higher success rate than the previous schemes in the batch verification phase. It improves batch signature verification performance by looking up signatures in a Cuckoo Filter before going through the expensive verification process. The paper also demonstrates through experiments that Cuckoo Filter outperforms Bloom Filter in performance measures.

3.6.2 Alleviation of DDoS Attack

DDoS threats have grown in sophistication and severity in recent times. Existing solutions to mitigate DDoS attacks employ packet filtering at a single layer (typically in firewall) when packets are transmitted to the server. However, this single-layer filtering is not free from errors in detecting packets targeted for attacks. This research [57] proposes a multi-layer scheme to filter packets. One of the layer uses a Cuckoo Filter to lower packet filtering overhead, and thus improves performance of packet filtering. The paper also demonstrates through experiments that the proposed scheme is able to better identify DDoS packets.

3.6.3 Deep Packet Inspection

Nowadays, Internet Service Providers (ISPs) have been depending on Deep Packet Inspection (DPI) approaches, which are the most precise techniques for traffic identification and classification. However, constructing high performance DPI approaches imposes a vigilant and an in-depth computing system design because of the requirements of high memory and processing power. Existing solutions use Bloom filters as a matching check tool in DPI approaches. This research [58] proposes replacing Bloom Filter with Cuckoo Filter to improve performance of Deep Packet Inspection. The paper demonstrates through experiments that Cuckoo Filter outperforms Bloom Filter in performance measures.

3.7 Cuckoo Filter in Big Data

After the introduction of Cuckoo Filter, recently, several research works have used Cuckoo Filter to speed up the lookup process in the areas of Networking and Security. However, there have not been significant researches utilizing Cuckoo Filter in the area of Big Data. This section briefly discusses the few researches that utilized Cuckoo Filter in the areas somewhat related to Big Data.

3.7.1 Privacy-Aware Search Over Encrypted Cloud Data

Many organizations and individual users are out-sourcing their information which includes sensitive data into the cloud. To deal with the potential risks of privacy exposure, such data is typically encrypted before being outsourced but users would like to conduct keyword-based searches. Traditional searchable encryption techniques are overly restrictive for they only allow exact keyword search. Thus, fuzzy keyword search is needed to deal with typos in users' search strings.

The research [59] presents a Cuckoo Filter Based Private Keyword Search Scheme (CFPKS) to provide privacy-aware keyword search over encrypted data. This CFPKS scheme uses a bed-tree structure-based index to boost search efficiency, a wildcard approach to support fuzzy keyword search, and a Cuckoo-filter to improve search accuracy and storage efficiency.

The research demonstrates the effectiveness and efficiency of the scheme using a large ACM publication dataset consisting of 0.1M documents. It also demonstrates through experiments that Cuckoo Filter outperforms Bloom Filter in performance measures.

3.7.2 SlimDB: A Space-Efficient Key-Value Storage Engine for Semi-Sorted Data

Modern key-value stores often use write-optimized indexes and compact in-memory indexes to speed up read and write performance. One popular write-optimized index is the Log-structured merge-tree (LSM-tree) which provides indexed access to write-intensive data. It has been increasingly used as a storage backbone for many services, including file system metadata management, graph processing engines, and machine learning feature storage engines. Existing LSM-tree implementations often exhibit high write amplifications caused by compaction, and lack optimizations to maximize read performance on solid-state disks.

This research [60] introduces Multi-Level Cuckoo Filters to improve performance by reducing number of disk reads required to lookup the LSM-tree index. Multi-Level ensures that multiple entries having the same fingerprint do not create collisions in the Cuckoo Filter.

The paper presents experiment results that show that by applying these design techniques, the new implementation of a key-value store can be two to three times faster, use less memory

to cache metadata indices, and show lower tail latency in read operations compared to popular LSM-tree implementations such as LevelDB and RocksDB.

3.7.3 Dynamic Cuckoo Filter

Many applications commonly involve a highly dynamic set with members joining and leaving dynamically and with an unpredictable size of the set. For an example, in a Web Cache Proxy, the cached set of web pages is frequently updated according to the cache replacement strategies. Large dynamic sets require that the capacities of structures support flexible extension or reduction and reliable delete operation. The deletion of any element of the set will not affect the accuracy of the membership testing of other elements in the set. Existing techniques for approximate set representation, e.g., the cuckoo filter, the Bloom filter and its variants cannot meet both the requirements of a dynamic set.

This research [49] proposes Dynamic Cuckoo Filter that supports reliable delete operation and elastic capacity for dynamic set representation and membership testing.

The paper demonstrates through experiments that compared to the existing state-of-the-art designs, Dynamic Cuckoo Filter achieves 75% reduction in memory cost, 50% improvement in construction speed, and 80% improvement in speed of membership query. The paper also shows through experiments that Dynamic Cuckoo Filter outperforms Dynamic Bloom Filter.

3.8 Summary

In this chapter, we have discussed contemporary research works on improving query performance in Big Data systems. The next chapter will discuss one of our proposed schemes to improve query execution performance in Big Data.

Chapter 4

Improving Query Execution Performance in Big Data using Cuckoo Filter

Big Data systems that offer high-performance query execution service usually utilize Bloom Filters along with *Eventual Consistency* model for performance improvement. 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*. 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.

We propose a scheme to improve performance of lookup queries after data deletion by replacing Bloom Filter with Cuckoo Filter that allows deletion of entries from within it. When rows are deleted, we delete the corresponding row keys from the Cuckoo Filter. Subsequent queries that try to look up those deleted keys will find that the filter does not contain the keys and hence will be able to avoid costly disk access. There are a few challenges in implementing this scheme, which we also address properly. We then show that the lookup query after data deletion can improve performance by up to 100% for each deleted record using our proposed scheme.

We also show that our scheme does not cause performance degradation as a side effect for any lookup or insertion queries whatsoever beyond the case of lookup queries after deletion.

4.1 Tombstoning Technique for Data Deletion and Its Limitations

Most of the Big Data systems that embrace the *Eventual Consistency* model to improve read/write performance employ a technique called *Tombstoning* for data deletion. Deletions are actually considered as updates and the data to be deleted is instead marked with a deletion timestamp, called a *tombstone*. After a certain period of time, regularly, disk data is compacted and during the compaction, tombstoned data is excluded from the compacted data, thus effectively deleting it.

Now, deletion is not supported by Bloom Filter. So Big Data systems that utilize Bloom Filters cannot delete entry from Bloom Filter when data is deleted from storage. That is why those systems update the row to be deleted with a tombstone timestamp for the time being. After a certain time interval, the storage data is compacted and Bloom Filter is rebuilt, effectively deleting the data from both the storage and filter. However, the time interval for data compaction is typically very long as the compaction process itself takes a lot of CPU resource and degrades the performance of the database system during its execution due to excessive disk read/write operation.

This technique has the disadvantage of lookup performance degradation after data deletion and before compaction. Because, in this case, data is actually read from the disk and then identified as tombstoned and removed from the final result. This results in executing a costly operation of disk access which is redundant. In this chapter, we propose a technique to improve the lookup performance in this case.

4.2 Scope of Query Performance Optimization for Data Deletion

There are three possible granular entities that can be deleted in a Big Data system - schema/table, row, column. We discuss each of these below and point out that the scope of query optimization lies in the case of row deletion only.

- **Deleting schema or table:** Dropping an entire table or schema actually removes an entry from the system table and the dropping is carried out immediately by the big data systems due to it having no side effects. Hence, the issue of query optimization does not arise in case of schema/table deletions.
- **Deleting columns in a row:** This is conceptually similar to setting `null` as the value of a column. This is rather an update operation than deletion and the row actually resides on disk. As the row is not really *deleted*, deletion of columns has no impact on query performance in any way.
- **Deleting entire rows:** Among the three different entities for deletion, row deletion is the most common case and it is in this case where there is scope to improve query performance. In this paper we propose a scheme to improve the lookup query performance after row deletion.

4.3 Proposed Method

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 encountered implementing the proposed scheme and discuss solutions to overcome those.

4.3.1 Query Performance Degradation in Case of Data Lookup After Deletion in Existing Big Data Systems

Most of the Big Data systems utilize the *Eventual Consistency* model to improve read/write performance employs the *Tombstoning* technique for data deletion. In this technique, deletions are actually considered as updates and the data to be deleted is instead marked with a deletion timestamp, called a *tombstone*. After a certain period of time, regularly, disk data is compacted and during the compaction, tombstoned data are excluded from the compact data, thus effectively deleting it. The compaction time interval is commonly a few days, because compaction is a very expensive operation and regular compaction would add significant overhead in terms of CPU usage and storage data processing. Again, many of these Big Data systems utilize *Bloom Filter*, a probabilistic data structure that can deterministically say if an item *does not* exist in it, but cannot surely confirm if an item *does* exist in it. Bloom Filter is utilized by the databases to determine if a particular row of data exists in disk storage against a row key, thus avoiding costly disk access in case the filter confirms that the data against a row key cannot be found from storage. 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 contains 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 that the row data has been marked with a tombstone and hence removes the row from the resultant set of rows to

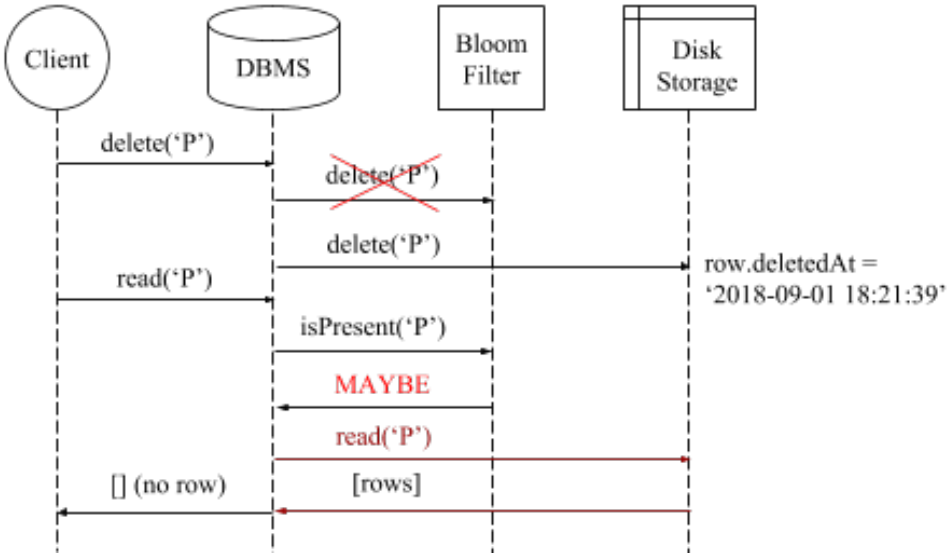


Figure 4.1: Query performance degradation in case of data lookup after deletion, due to Bloom Filter not supporting deletion of elements from within it.

be returned to the client. This unnecessary storage access increases query execution time significantly and hence degrades query performance.

Figure 4.1 depicts the above scenario.

4.3.2 Proposed Scheme to Improve Performance of Lookup Query After Deletion

In this paper, we propose a scheme that replaces Bloom Filter with Cuckoo Filter that allows deletion from the filter. So when a row is deleted, the corresponding row key is also deleted from the filter. Consequently, for forthcoming queries that inquire for data against that deleted row key, the filter will deterministically say that the data against the key does not exist in disk (or, in this case, it existed, but now has been deleted); hence the significant cost of disk access time can be avoided, making the query performance a lot better. Following is an illustrative scenario of how the proposed scheme works.

1. A client makes a row deletion query. The query is executed exactly the way executed by the current system, that is, data in disk is marked with a tombstone. We do not propose

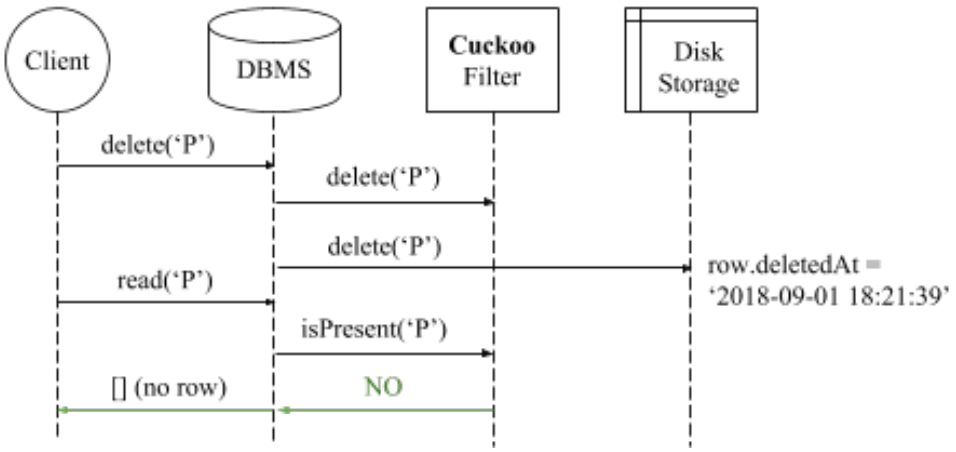


Figure 4.2: Cuckoo Filter improves performance by supporting deletion operation.

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 that has been 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-save way and creates minimal overhead, we propose using *Adaptive Cuckoo Filter*, a variant of Cuckoo Filter that removes keys resulting in false positives from itself. Section 4.3.3.2 expands upon the issue in details.

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, thus improving performance than current systems.

Figure 4.2 depicts how Cuckoo Filter improves performance by supporting deletion operation.

Algorithm 2 outlines the steps for deletion of rows as described in the aforementioned proposed scheme. Similarly, Algorithm 3 outlines the steps for lookup query execution against a partition key using the proposed scheme.

Through experiments, we show that the proposed scheme can improve performance of lookup queries after deletion for up to 100%. We also show through extensive experiments that our

Algorithm 2: Deletion in proposed scheme (`DeleteRows(`$partitionKey$`)`).

```
update rows with tombstone marker;
CuckooFilter.delete(partitionKey);
CuckooFilter.save();
```

Algorithm 3: Lookup in proposed scheme (`Lookup(`$partitionKey$`)`).

```
if CuckooFilter.mayContain(partitionKey) then
    rows = read rows from disk against partitionKey;
    return rows;
else
    return; // no rows
end
```

scheme does not cause performance degradation as a side effect for any lookup or insertion queries whatsoever beyond the case of lookup queries after deletion.

4.3.3 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 with very minimal overhead. In this section we discuss the challenges and propose solutions to overcome those.

4.3.3.1 When consistency level requirement is more than one nodes

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. For example, in case of an ecommerce application, a shopper's notifications, purchase history etc. can be dealt with weak consistency; but if a shopper loads money into the wallet of the application and then sees the available amount in wallet does not reflect the loaded amount, it could be an important issue considered by the ecommerce company and hence the company may require strong

consistency on wallet amount data.

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 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 the restored node is the latest one, which is clearly wrong. The system derives this conclusion based on the fact that the system compares 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 (Figure 4.3).

This can be handled properly by using a modified version of Cuckoo Filter where it will not only respond 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 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 in disk is marked with a tombstone.

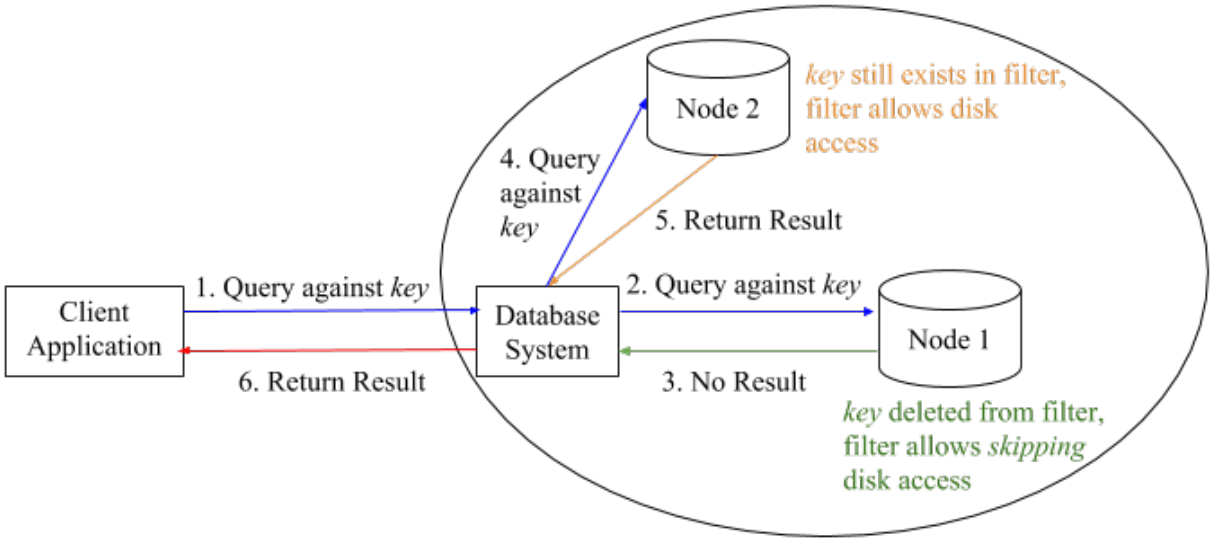


Figure 4.3: Failure of Big Data system in inferring correct result with our proposed scheme applied, in the case where Consistency Level requirement > 1.

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 contains 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 (Figure 4.4).

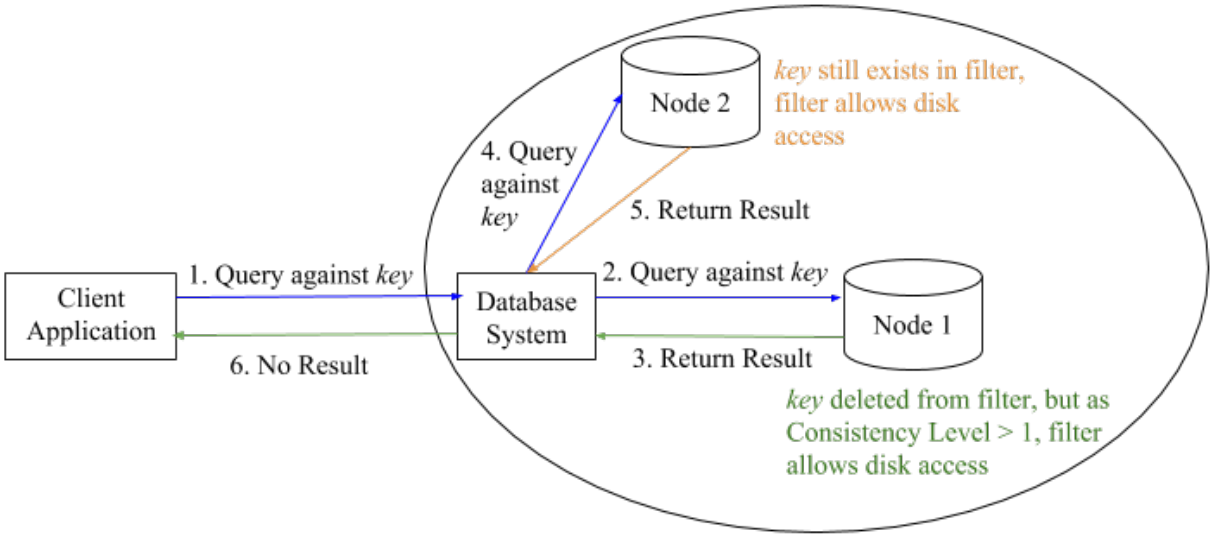


Figure 4.4: Modifying the behavior of Cuckoo Filter makes sure the Big Data system infers correct result with our proposed scheme applied, in the case where Consistency Level requirement > 1.

Figure 4.5 depicts the aforementioned steps.

Algorithm 4 outlines the steps for deletion of rows as described in the aforementioned modified proposed scheme. Similarly, Algorithm 5 outlines the steps for lookup query execution against a partition key using the modified proposed scheme.

4.3.3.2 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 (also depicted in Figure).

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.
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.

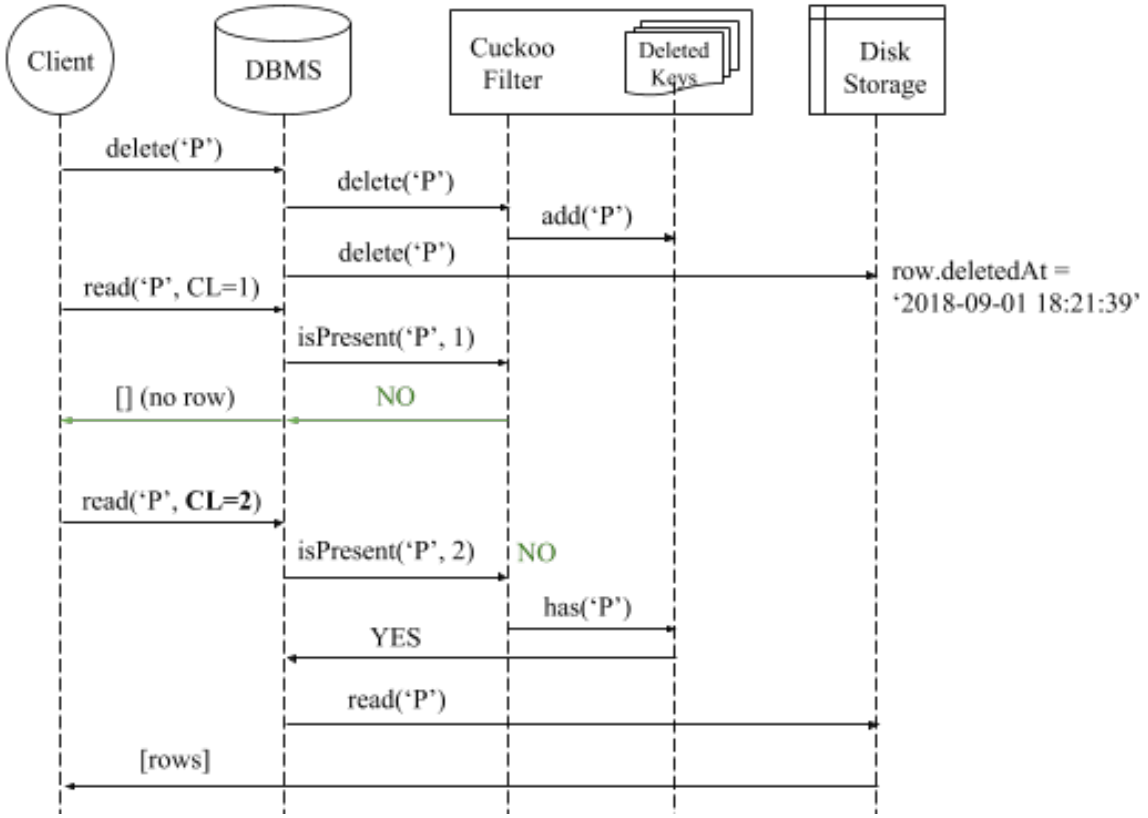


Figure 4.5: Modification of Cuckoo Filter to handle consistency level requirement.

Algorithm 4: Deletion in modified proposed scheme (`DeleteRow(`$partitionKey$`)`).

update row with tombstone marker;
CuckooFilter.delete($partitionKey$);
CuckooFilter.$deletedKeys$.add($partitionKey$);
CuckooFilter.save();

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 [48], 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.

Algorithm 5: Lookup in modified proposed scheme (`Lookup(`$partitionKey$`,` $consistencyLevel$`)`).

if CuckooFilter.mayContain($partitionKey$) **then**
  $rows$ = read rows from disk against $partitionKey$;
  return $rows$;
else if $consistencyLevel > 1$ and CuckooFilter.$deletedKeys$.contains($partitionKey$) **then**
  $rows$ = read rows from disk against $partitionKey$;
  return $rows$;
else
  return; *// no rows*
end

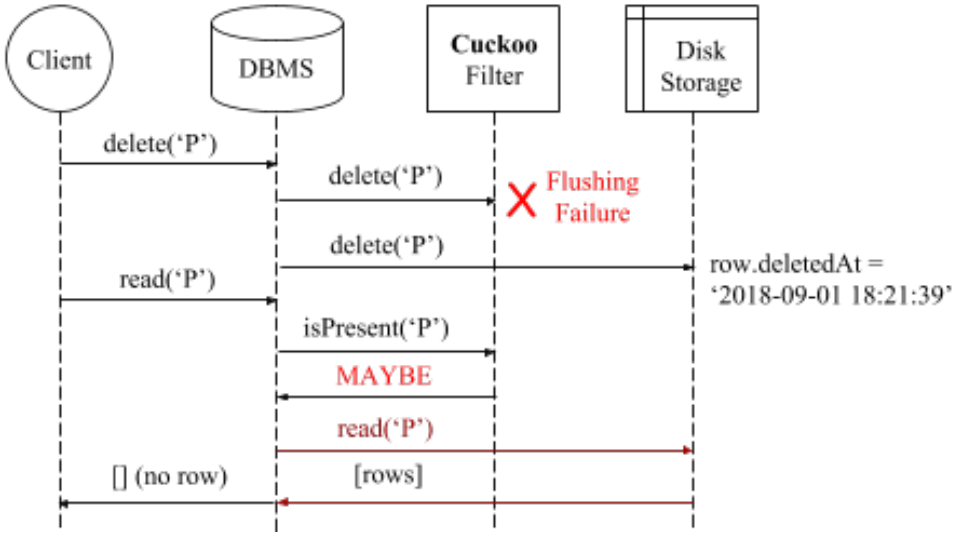


Figure 4.6: Performance degradation in case flushing a modified filter fails.

4.4 Experimentation

4.4.1 Experiment 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 cases where data does not exist on disk. Examples of popular Big Data systems incorporating these features include Google BigTable [18], Apache HBase [21] and Apache Cassandra [19]. 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 [26]. Cassandra offers robust support for clusters spanning multiple datacenters [27] with asynchronous masterless replication allowing low latency operations for all clients [26].

2. It is the most popular Column-Store Big Data database as of July 2018 according to DB-Engines ranking [29].

3. Cassandra is open-source [28].

4. The source code base of Cassandra is organized and clean. Also, querying the database is very simple 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 [61] and plugged an open-source implementation of Cuckoo Filter [62] into it. We made necessary changes according to our proposed methodology.

We compiled the source code and ran our experiments in two environments - a simulated Big Data environment set up on a local machine and a real Big Data environment set up on Amazon Web Services (AWS) cloud computing system. The local machine was an Apple Macbook Pro workstation having Quad Core 2.3 GHz Intel Core i7 processor with 6MB L3 Cache and 8 GB 1600 MHz DDR3 RAM. The AWS node 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 3,500 Mbps dedicated EBS bandwidth. The experiments were executed from a same-configuration node in the same availability zone (us-east-2a).

As for a real-world experiment dataset, for the case of simulated environment, we chose to use US Airports flight statistics dataset (10M records, 2GB total data size) that is available publicly [63]. For the real environment, we chose to use Amazon Customer Reviews dataset (136M records, 50GB total data size) that is also available publicly [64]. We inserted the

Table 4.1: Performance comparison summary between Cuckoo Filter and Bloom Filter.

Aspect	**Cuckoo Filter**	**Bloom Filter**
Lookup after deletion	Improves for up to 100%	No improvement
Lookup in general	Similar	Similar
Insertion in general	Similar	Similar

dataset into our Cassandra instances using the built-in tool for data manipulation named *CQLSH* (*Cassandra Query Language Shell*).

We developed a program in Java [65] to connect to our Cassandra instance, execute queries and collect execution time.

4.4.2 Experimental Results

In Section 4.4.2.1, we evaluate the performance of our proposed method. To show that the core change proposed in our method, 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 Sections 4.4.2.2 and 4.4.2.3.

Table 4.1 summarizes the performance comparison between Cuckoo Filter and Bloom Filter as can be seen from the outcome of the experiments.

4.4.2.1 Lookup performance after deletion

From Figures 4.7 and 4.8 we see that allowing deletion in filter improves query execution performance significantly (in simulated environment for 97% and in AWS environment for 99.96%) for querying data that have been deleted. This leads to the conclusion that the more volume of data can be skipped from reading from disk storage, the faster the query executes. We have also done experiment that proves this conclusion is correct. Figures 4.9 and 4.10 show 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

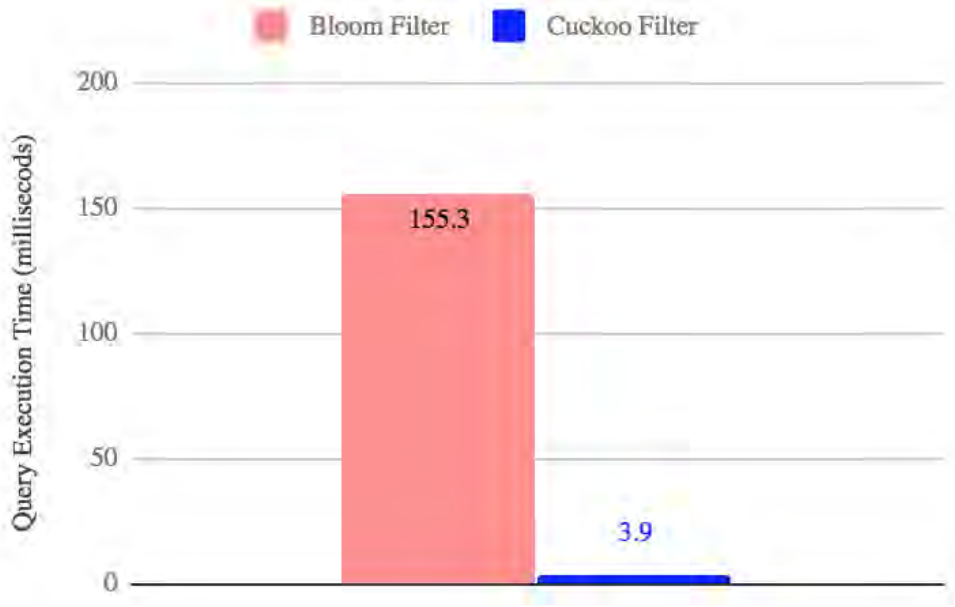


Figure 4.7: Lookup performance after deletion in simulated environment. Cuckoo Filter improves performance for 97%. Each value is the average of 10 runs.

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% 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.

In the AWS environment, we also experimented with lookup after deletion performance against varying data size, and from Figure 4.11 it can be seen that the more data is deleted, the more performance improvement Cuckoo filter achieves.

4.4.2.2 Lookup performance in general

We experimented for evaluating general lookup performance in two ways - by varying the amount of queries that return results, and by varying filter load (that is, how full a filter is). These two ways have been devised from the experiments conducted in the Cuckoo Filter paper [8].

Figures 4.12 and 4.12 show the comparison result for varying the fraction of queries that

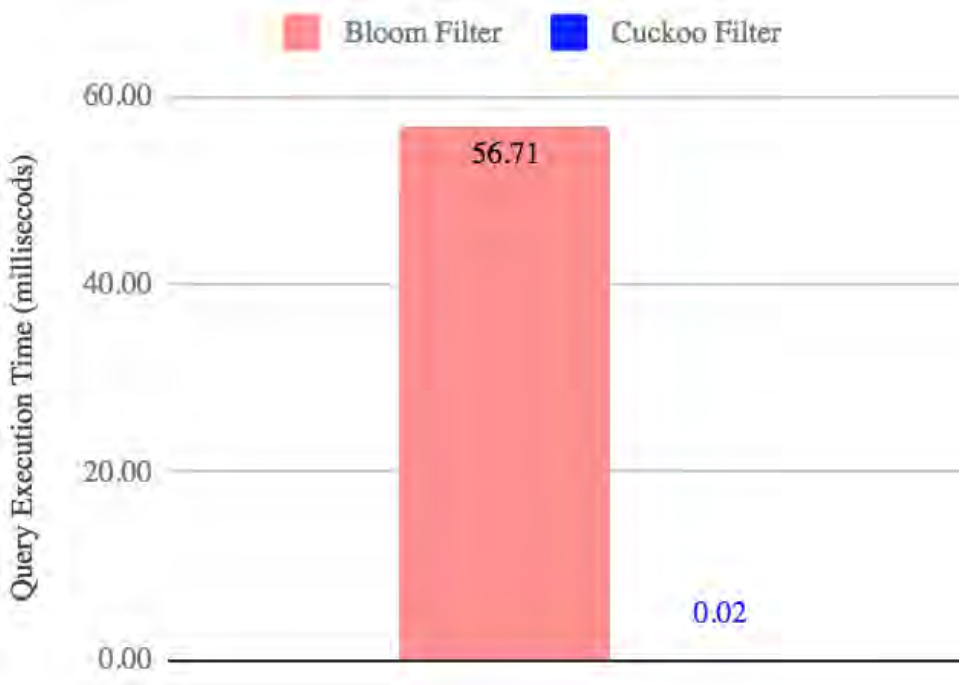


Figure 4.8: Lookup performance after deletion in AWS environment. Cuckoo Filter improves performance for 99.96%.

return positive results. Naturally, the more queries return positive results, the more data needs 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 resulting in slightly better execution time.

Figure 4.14 shows the result of comparison for varying the filter load, that is, how full the filter is. Typically the more the filter load is, the worse Bloom Filter performs, whereas Cuckoo Filter performance remain stable even if filter load reaches near 100%. However, the performance penalty is very minimal (in microseconds range) compared to overall query performance (in milliseconds range) and hence it is not reflected in the chart. However, the main purpose of the experiment is to show that Cuckoo Filter does not degrade query performance than when Bloom Filter is used, and the chart clearly shows that is true.

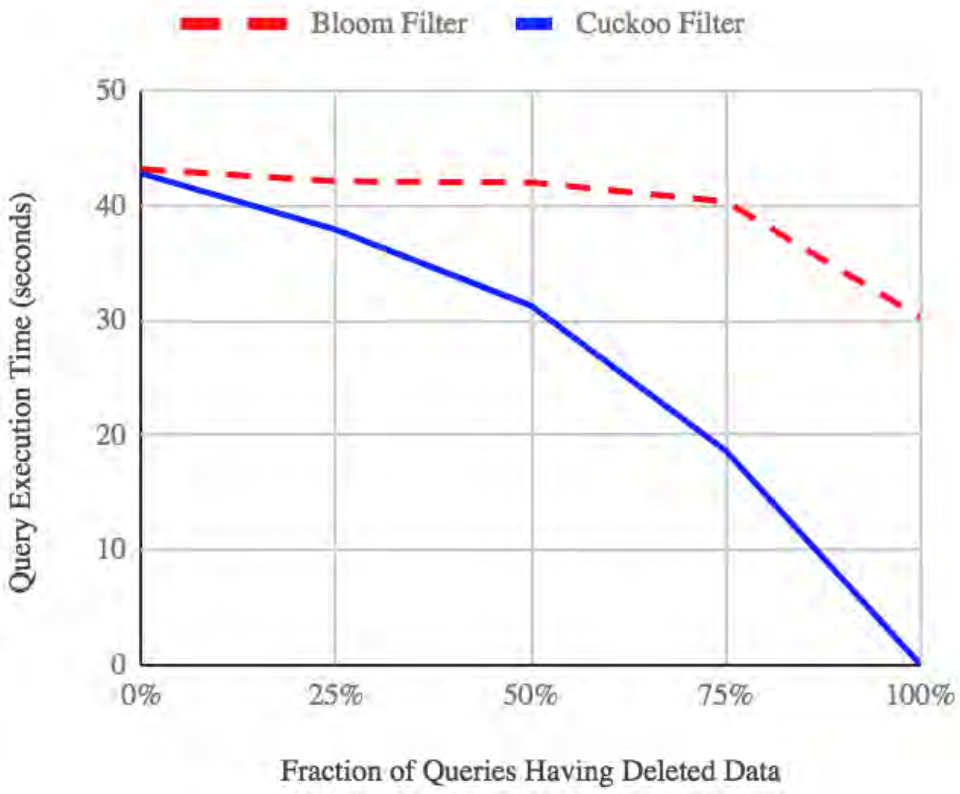


Figure 4.9: Lookup performance after deletion in simulated environment for varying percentage of queries returning deleted data. Cuckoo Filter improves performance for up to 100% depending on percentage of queries returning deleted data. Each value is the average of 10 runs.

4.4.2.3 Insertion performance

Figure 4.15 shows the result of our experiment with insertion query execution and it clearly shows that Cuckoo Filter performance is again equivalent to Bloom Filter, while being slightly better.

4.5 Summary

In this chapter, we have proposed a scheme to improve lookup query performance after data deletion in Big Data systems by replacing Bloom Filter with Cuckoo Filter. To handle some edge cases during implementation, we modified the Cuckoo Filter to include a list of deleted keys, so that queries requiring consistency level higher than 1 result in correct data. We also

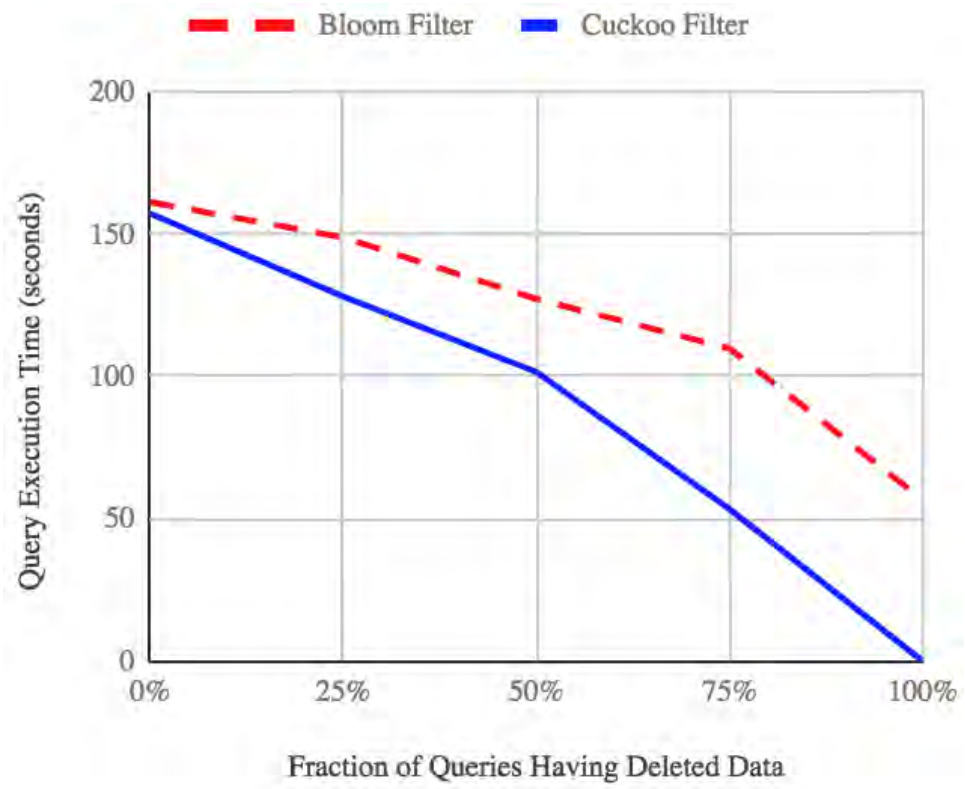


Figure 4.10: Lookup performance after deletion in AWS environment for varying percentage of queries returning deleted data. Cuckoo Filter improves performance for up to 100% depending on percentage of queries returning deleted data.

used a variation of Cuckoo Filter named *Adaptive Cuckoo Filter* that handles deletion of keys from within it in a robust way.

The next chapter discusses another scope for query performance improvement in distributed Big Data Systems.

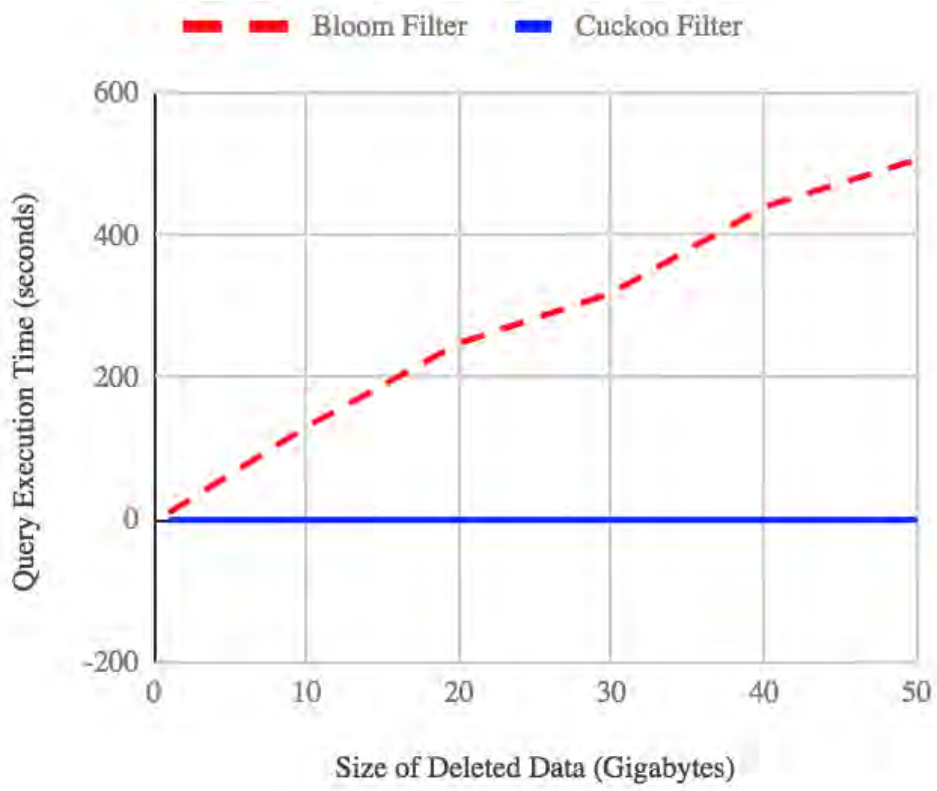


Figure 4.11: Lookup performance after deletion in AWS environment for varying data size. Cuckoo Filter improves performance for up to 100%.

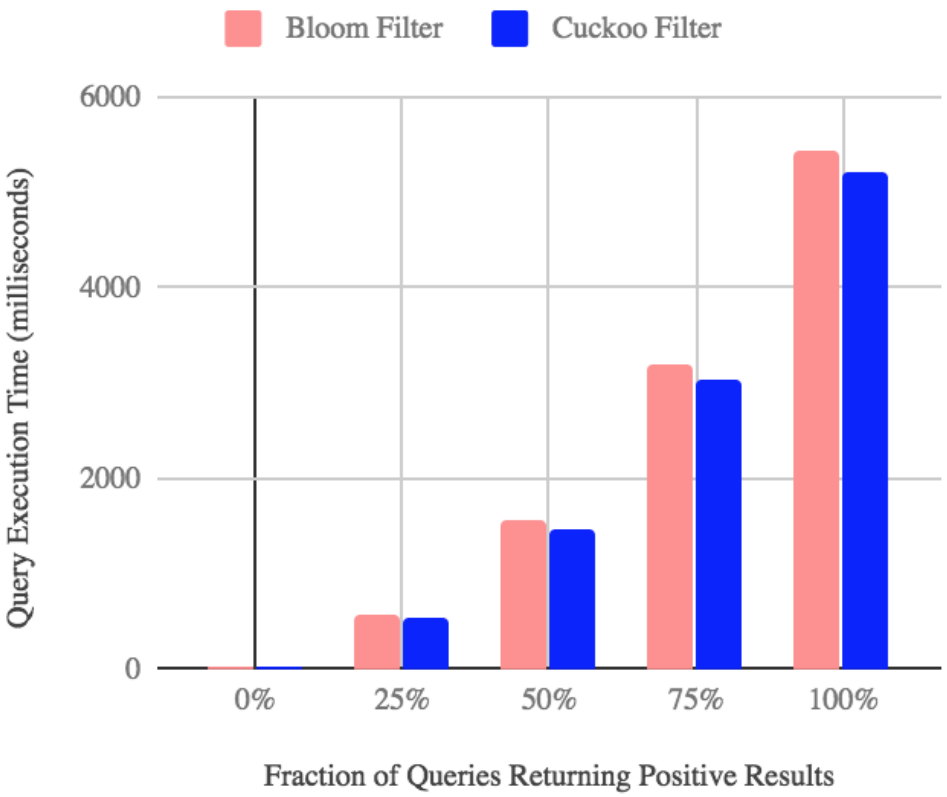


Figure 4.12: Lookup performance for varying query result positiveness (that is, percentage of queries returning positive results) in simulated environment. Performance of both Bloom Filter and Cuckoo Filter is very similar. Each value is the average of 10 runs.

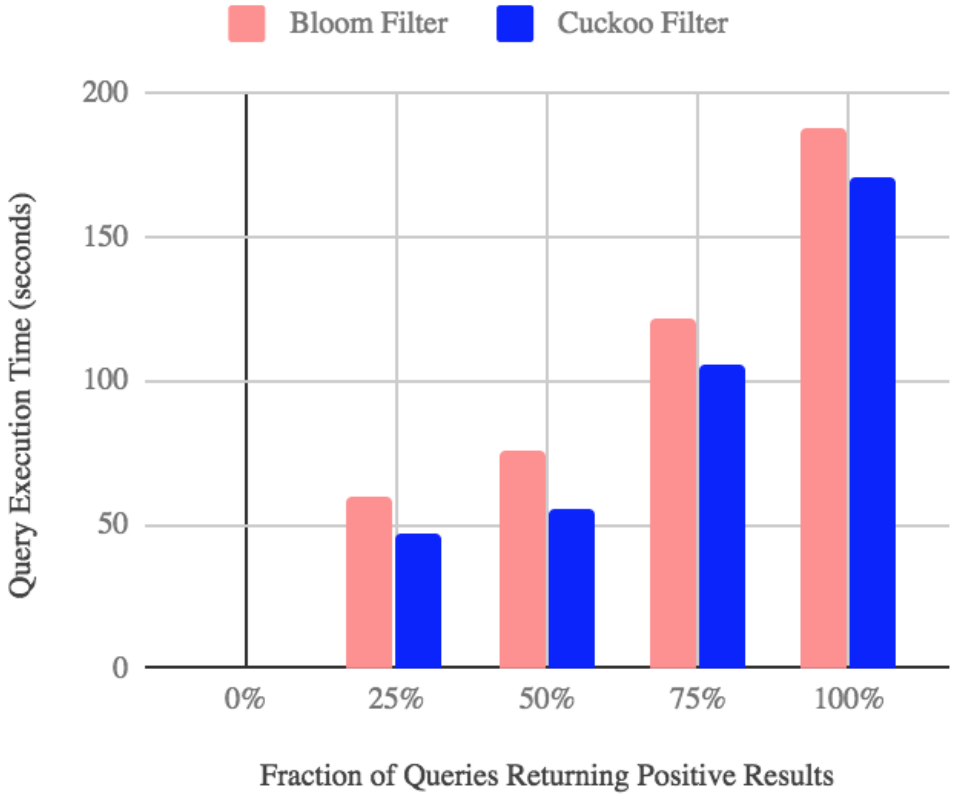


Figure 4.13: Lookup performance for varying query result positiveness (that is, percentage of queries returning positive results) in AWS environment. Performance of both Bloom Filter and Cuckoo Filter is very similar.

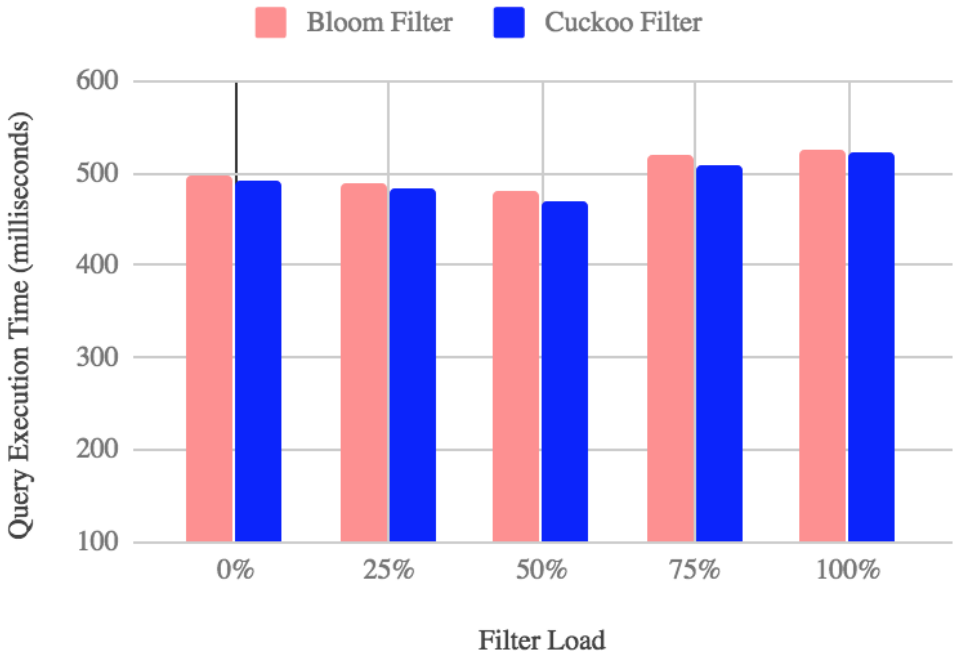


Figure 4.14: Lookup performance for varying filter load (that is, how full the filter is). Performance of both Bloom Filter and Cuckoo Filter is very similar. Each value is the average of 10 runs.

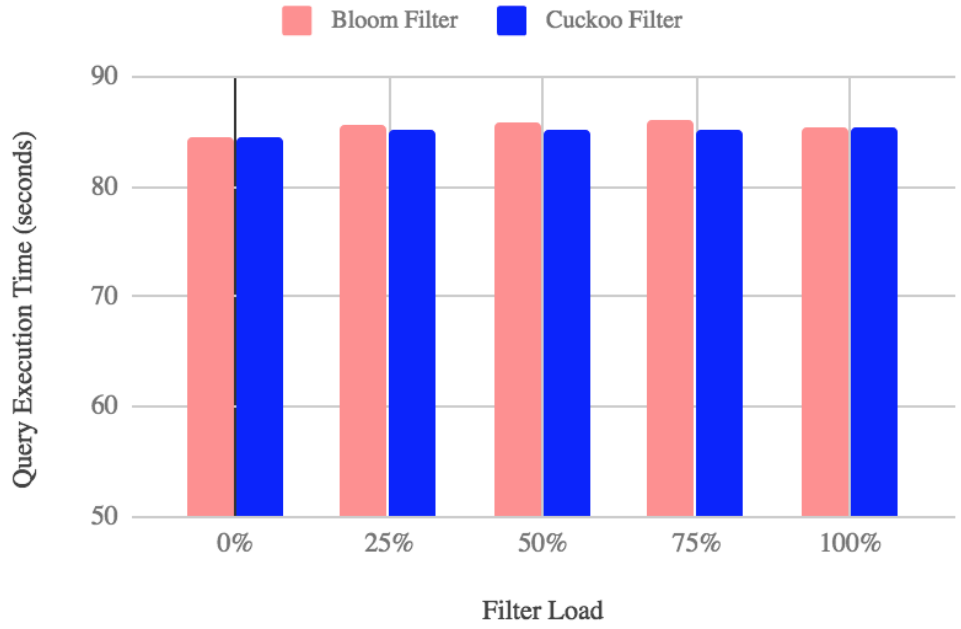


Figure 4.15: Insertion performance for varying filter load (that is, how full the filter is). Performance of both Bloom Filter and Cuckoo Filter is very similar. Each value is the average of 10 runs.

Chapter 5

Improving Query Execution Performance in Distributed Big Data Systems using Probabilistic Filters

Big Data is composed of several components and various researches proposed techniques to improve those components. Starting from efficient indexing and caching, techniques such as improved query execution plans and effective data partitioning have been explored by researchers. 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.

We introduce a probabilistic data structure called a *filter* that will store partition keys against which a node has data within it. Probabilistic filters have extremely low memory footprints that allow them to be passed around among nodes in a network without causing network traffic overhead. The nodes will sync the filters among themselves. A client connects to a node and executes lookup queries 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 otherwise

increase query execution time.

We have run several carefully-designed experiments in a popular Big Data database (Cassandra) with a real data set to evaluate our scheme and have shown that the performance of lookup queries improves for up to 100% in cases where data do not exist in remote nodes. The experiments cover practial 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 while introducing node filter adds a very low amount of data insertion and lookup overhead, it is ignorable for practical purposes.

5.1 Query Execution in a Big Data Cluster of Nodes and Its Limitation

We 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, 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 (Figure 5.1).

3. If, however, 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 (Figure 5.2). Note that in this case, even 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.

5.2 Proposed Method

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

1. We introduce a filter in each node for each database table, which we name *Node Filter*.

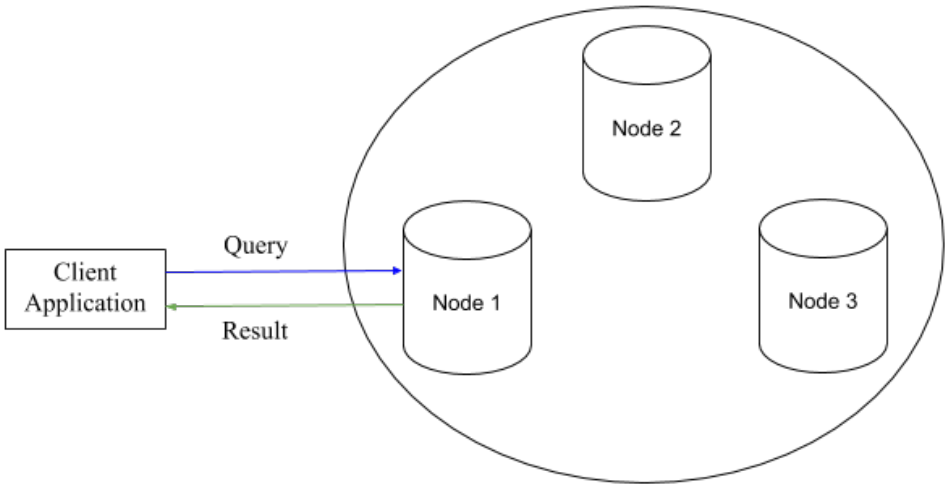


Figure 5.1: Query execution path when cluster node *owns* partition of data requested by client.

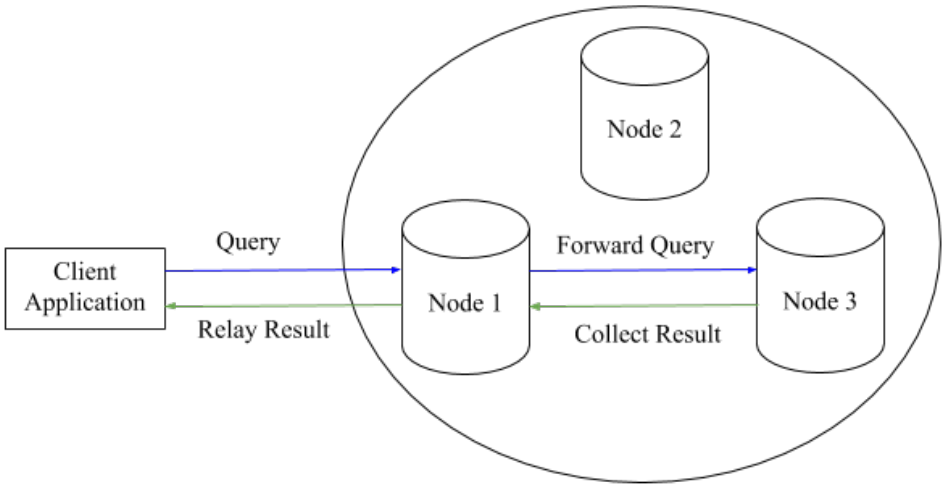


Figure 5.2: Query execution path when cluster node *does not own* partition of data requested by client.

2. For each table, the node filter contains all the keys against which data is stored in the table in that node.

3. Whenever there is a change against a key in the data stored in disk (i.e. a row with the partition key is inserted into or removed from the filter), the corresponding node filter is changed accordingly.

4. Each node will have a copy of the node filters in other nodes in the cluster. The copies will

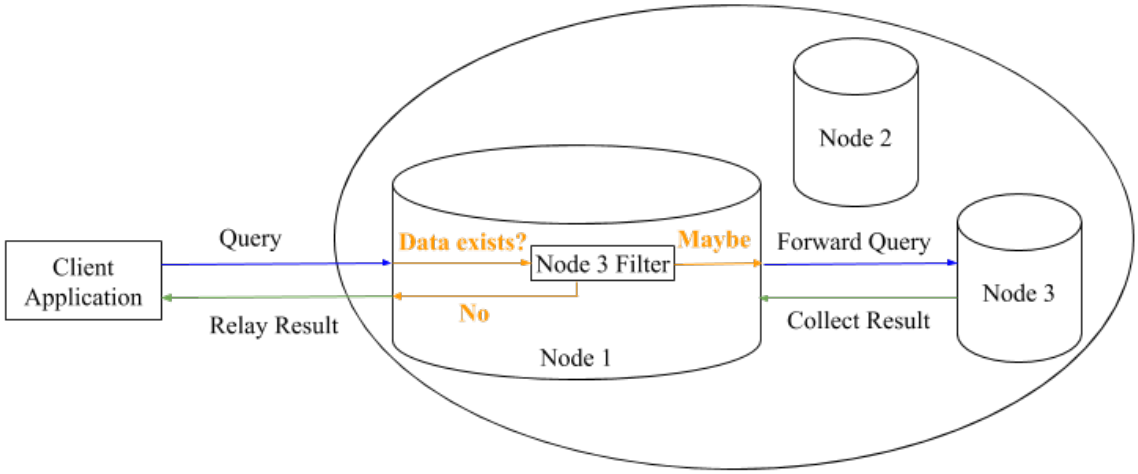


Figure 5.3: Query execution path when node filter is used. Query is relayed to remote node only if the node filter indicates existence of data in the target node.

be synchronized among the nodes as described in Section 5.2.2.1.

5. When a client connects to a node and the node finds out that another node owns the requested partition, it first looks up the partition key in the copy of the node filter it has of the target node.

6. If the node filter suggests that the remote node might contain data for the requested partition key, it proceeds to reach the target node to retrieve data as usual.

7. If, however, the node filter confirms that the remote node does not have data for the requested partition key, it can simply return empty result against that node instead of collecting the same result from the target node and relaying it to the client; thus saving network communication and improving query execution performance.

Figure 5.3 shows the query execution paths when this node filter is used.

Algorithm 6 outlines the steps a node filter takes when a partition key is looked up.

Algorithm 6: Node Filter lookup (`Lookup` ($partitionKey$)).

```
if partitionKey owned by local node then
    rows = read rows from local storage against partitionKey;
    return rows;
else
    if NodeFilter.mayContain(partitionKey) then
        rows = read rows from remote node against partitionKey;
        return rows;
    else
        return; // no rows
    end
end
```

5.2.1 Using Cuckoo Filter to Improve Lookup Performance After Data Deletion

Step 3 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 node 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.

5.2.2 Challenges and Issues in Implementing Node Filter

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 propose effective solutions.

5.2.2.1 Synchronizing node filters among nodes

The most effective way to synchronize node filters among nodes 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 (Figure 5.4). 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 (Figure 5.5) and peer-to-peer architecture (Figure 5.6). However, this solution is complex and has its own overhead. In case of implementation of this solution in peer-to-peer distributed architecture, what happens if a node recovers and before it gets a chance to update its node, it rather tries to sync its version of node filter with other nodes (Figure 5.7)? We now need to introduce a timestamp-based update operation to solve this issue, creating further 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

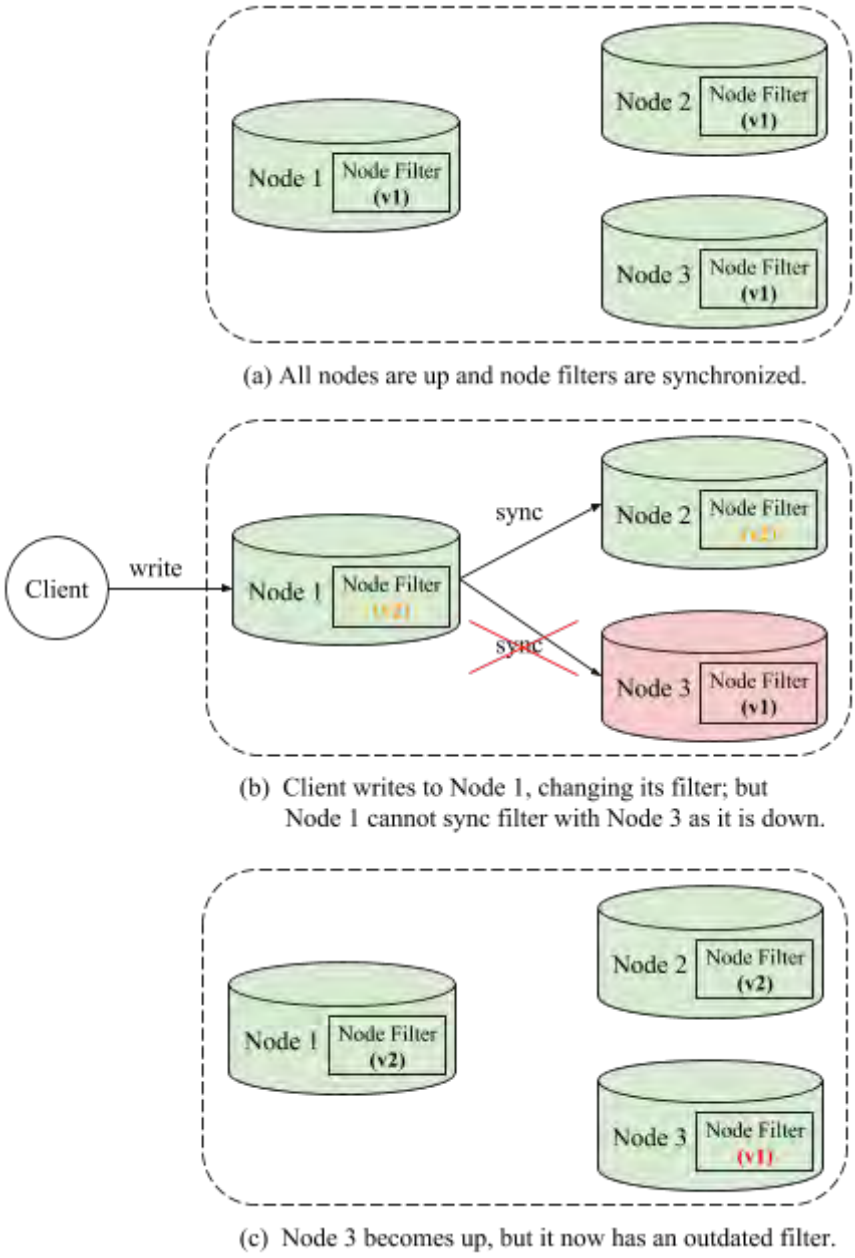


Figure 5.4: Node filter synchronization issue during failure of a node.

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.

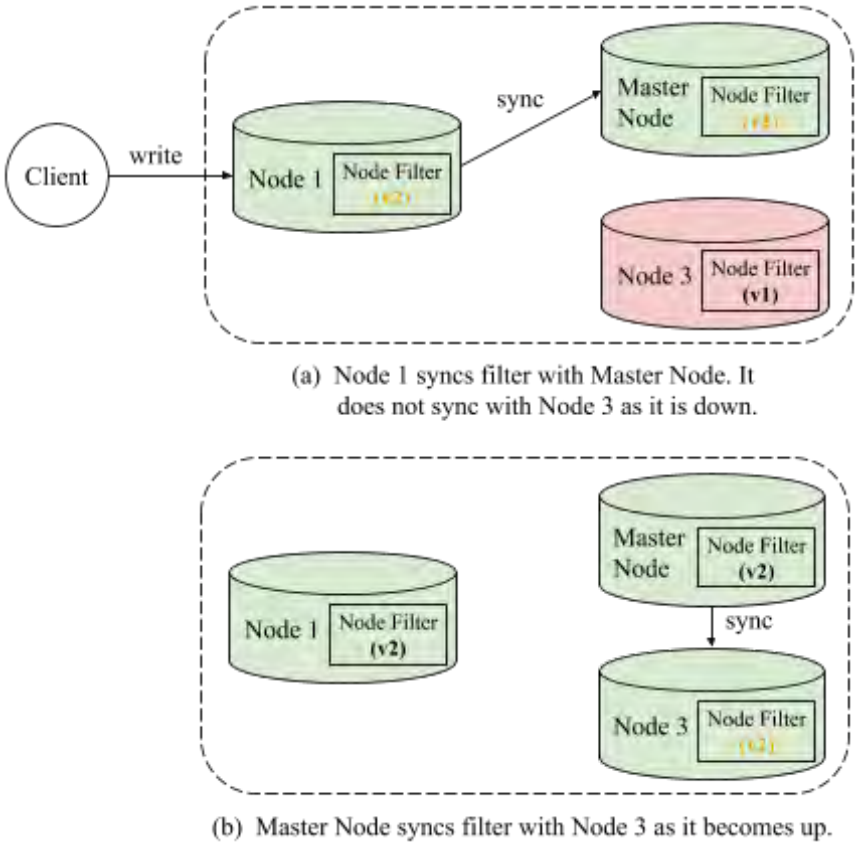


Figure 5.5: Proper way to synchronize node filter in master-client distributed architecture.

5.2.2.2 False-negatives within the time between 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 (Figure 5.8).

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

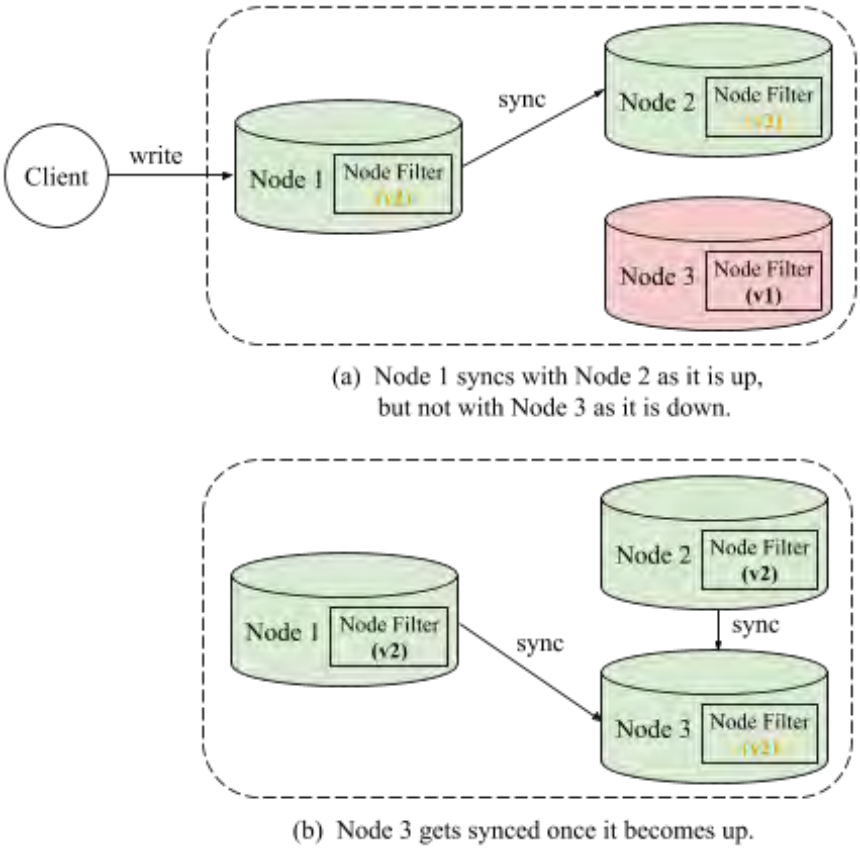


Figure 5.6: Proper way to synchronize node filter in peer-to-peer distributed architecture.

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.

5.2.2.3 Effect on general lookup and insertion query performance

Introducing Node Filter may be viewed as an extra overhead in query performance. However, the performance overhead is too minimal to be considered as noticeable, let alone being significant. The lookup and insertion performance of the filters are both $O(1)$. Moreover, each lookup

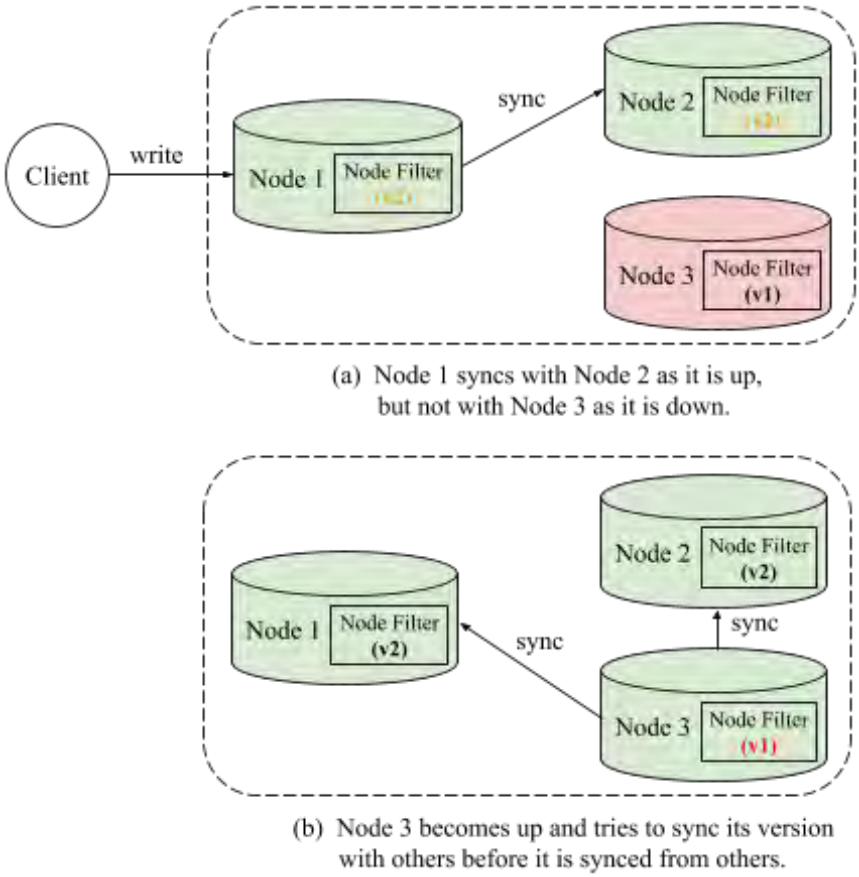


Figure 5.7: Issue in synchronizing node filter in peer-to-peer distributed architecture.

takes up to a few milliseconds, whereas practical Big Data query execution time ranges from seconds to several minutes. Therefore, the performance penalty of node filters is safely ignorable for practical purposes. We also show in our experiments that the benefit gain is much more significant than the negligible overhead caused by node filters.

5.3 Experimentation

5.3.1 Experiment Setup

Most Big Data systems currently in use have an efficient method for choosing a replica node that contains the requested data partition and has the lowest network latency from the source node. Most Big Data systems currently in use have such efficient lookup methods. Examples of popular

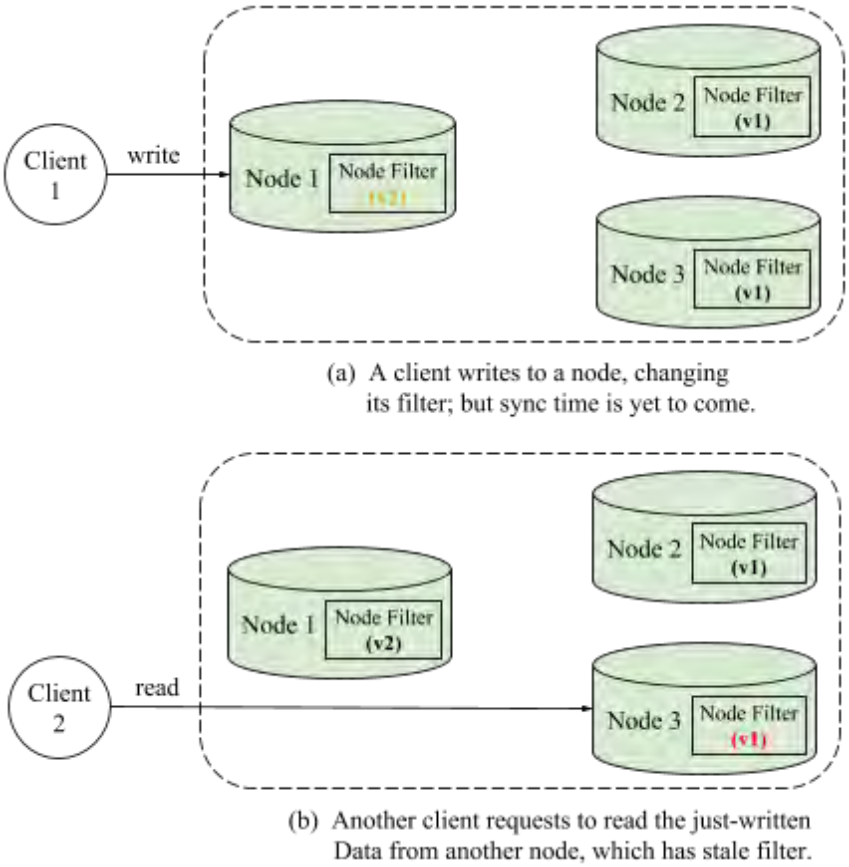


Figure 5.8: False-negatives within the time between filter update and synchronization

Big Data systems incorporating these features include DynamoDB [66], Apache Cassandra [19], Couchbase [67], Voldemort [68]. 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 [26]. Cassandra offers robust support for clusters spanning multiple datacenters [27] with asynchronous masterless replication allowing low latency operations for all clients [26].

2. It is the most popular Column-Store Big Data database as of July 2018 according to DB-Engines ranking [29].

3. Cassandra is open-source [28].

4. The source code base of Cassandra is organized and clean. Also, querying the database is very simple 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 [61] and plugged an open-source implementation of Cuckoo Filter [62] into it. We made necessary changes according to our proposed methodology.

We compiled the source code and ran our experiments in two environments - a simulated environment using regular workstations and a real environment using a Cassandra Cluster set up in Amazon Web Services (AWS). As for the simulated environment, we ran experiments in two Apple Macbook Pro workstations connected in a Local Area Network, each having Quad Core 2.3 GHz Intel Core i7 processor with 6MB L3 Cache and 8 GB 1600 MHz DDR3 RAM. The two workstations were set up as a two-node cluster. As for the real world environment, we set up a Cassandra Cluster of 6 nodes spread in three availability zones (us-east-2a, us-east-2b, us-east-2c) in the US-East (Ohio) region. Each node was set up on 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 3,500 Mbps dedicated EBS bandwidth. 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, for the case of simulated environment, we chose to use US Airports flight statistics dataset (10M records, 2GB total data size) that is available publicly [63]. For the real environment, we chose to use Amazon Customer Reviews dataset (136M records, 50GB total data size) that is also available publicly [64]. We inserted the dataset into our Cassandra instance using the built-in tool for data manipulation named *CQLSH* (*Cassandra Query Language Shell*).

We developed a program in Java [65] to connect to one of the Cassandra nodes in the cluster, execute queries and collect execution time.

5.3.2 Experiment Results

In Section 5.3.2.1, we evaluate the performance of our proposed method, comparing among the default implementation and Node Filters utilizing Bloom and Cuckoo Filters. We show in Section 5.3.2.2 that Cuckoo Filter dramatically improves performance over Bloom Filter in case of lookup query after data deletion. In Section 5.3.2.3, we evaluate the performance overhead of our proposed method when 100% queries result in positive data and conclude that the performance penalty is minimal. Similarly, in Section 5.3.2.4 we show that the insertion performance penalty is ignoreable. Finally, in Section 5.3.2.5 we look at the filters from the perspective of false positive rate and its effect and show that the less false positive rate we want to achieve, the more filter size becomes.

Table 5.1 summarizes the performance comparison of Node Bloom Filter and Node Cuckoo Filter in how they compare to the default implementation; as can be seen from the outcome of the experiments.

Table 5.1: Summary of how Node Bloom Filter and Node Cuckoo Filter compares with default implementation.

Aspect	**Node Bloom Filter**	**Node Cuckoo Filter**
Lookup performance when remote node does not contain data against queried partition key	Improves for up to 100%	Improves for up to 100%
Lookup performance after data deletion	Similar to default implementation	Improves for up to 100%
Lookup performance when remote nodes contain data against all queried partition keys	Similar to default implementation	Similar to default implementation
Insertion performance	Similar to default implementation	Similar to default implementation

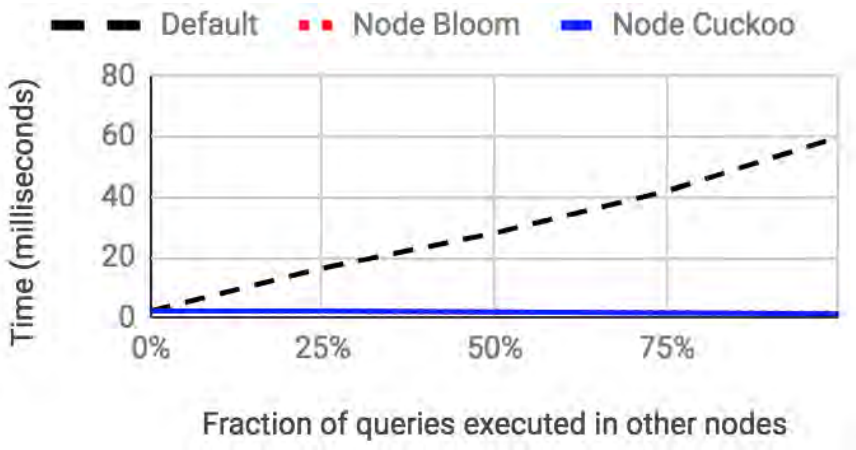


Figure 5.9: Lookup performance in simulated environment by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns negative results. Node filter improves query performance for up to 100%, depending on the fraction of queries executed in remote node. The performance of Node Bloom and Node Cuckoo is very similar and hence overlaps. Each value is the average of 10 runs.

5.3.2.1 Lookup performance when remote node does 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 node. We maintain that none of the queries that are executed in the connected node or the remote node results in any row. Figures 5.9 and 5.10 show 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 implementations improves performance for up to 100%.

We ran another experiment from a different perspective - measuring performance by varying a fraction of queries that look up partition keys against which the remote node does not contain data. Here we maintained that partition keys in all the queries would require looking up data from the remote node. Figures 5.11 and 5.12 show the result. We can see that in case of all queries returning positive results, our proposed scheme has a very slight performance penalty of 2%, which is quite ignorable as it is in the milliseconds range and would not cause much latency when querying huge amount of data that results in seconds of query execution time. However, the more fraction of queries returns empty results, the better the performance becomes; up to the

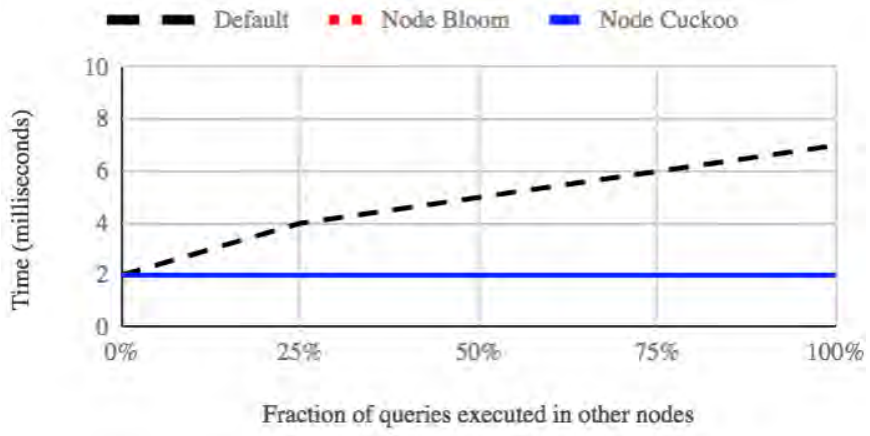


Figure 5.10: Lookup performance in AWS environment by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns negative results. Node filter improves query performance for up to 100%, depending on the fraction of queries executed in remote node. The performance of Node Bloom and Node Cuckoo is very similar and hence overlaps.

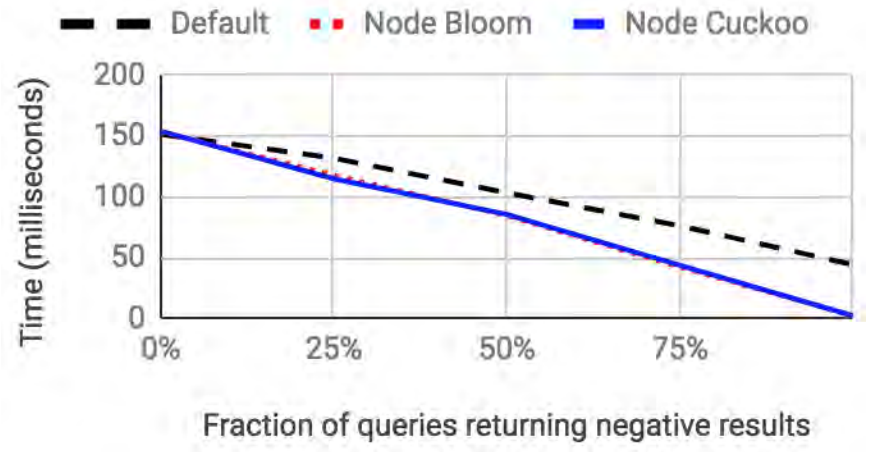


Figure 5.11: Lookup performance in simulated environment by varying fraction of queries returning negative results, while all queries are relayed to remote node for execution. Node filter improves query performance for up to 100%, depending on the fraction of queries returning negative results. The performance of Node Bloom and Node Cuckoo is very similar and hence overlaps. Each value is the average of 10 runs.

point that in case of 100% queries resulting in negative results, the performance improvement is 100%.

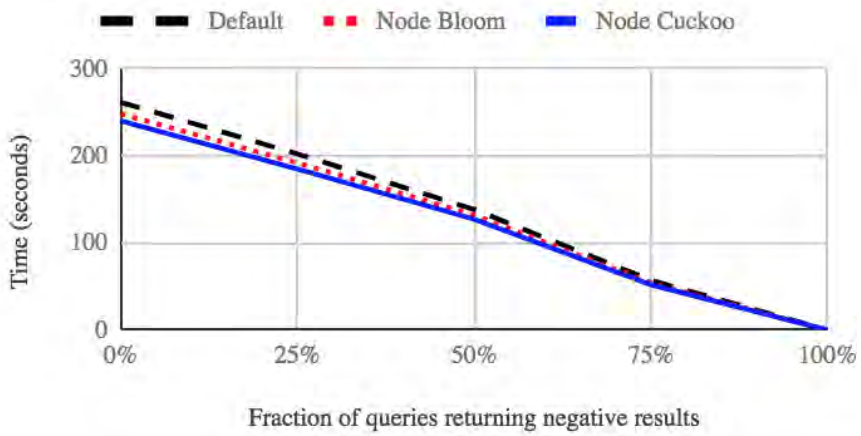


Figure 5.12: Lookup performance in AWS environment by varying fraction of queries returning negative results, while all queries are relayed to remote node for execution. Node filter improves query performance for up to 100%, depending on the fraction of queries returning negative results. The performance of Node Bloom and Node Cuckoo is very similar and hence overlaps.

5.3.2.2 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 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%, as can be seen from Figures 5.13 and 5.14. In the AWS environment, we also experimented with lookup after deletion performance against varying data size, and from Figure 5.15 it can be seen that the more data is deleted, the more performance improvement Node Cuckoo scheme achieves.

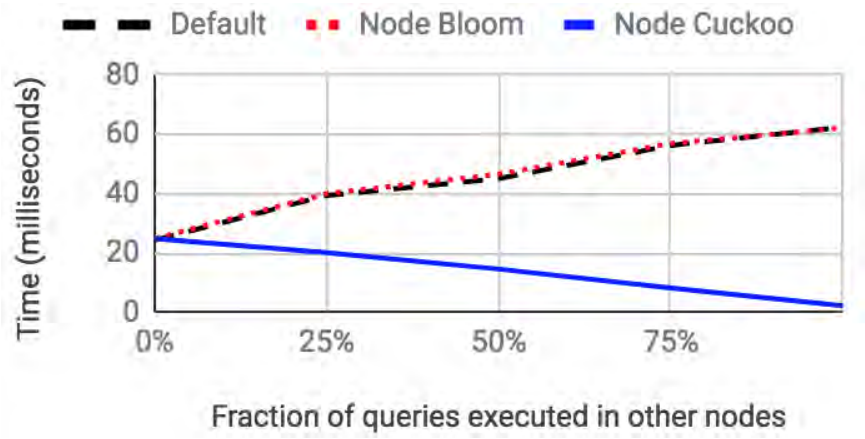


Figure 5.13: Lookup performance in simulated environment after data deletion, by varying fraction of queries retrieving rows from remote node. All the queries result in deleted data. Node Cuckoo filter improves lookup performance for up to 100%, depending on the fraction of queries executed in other nodes. Each value is the average of 10 runs.

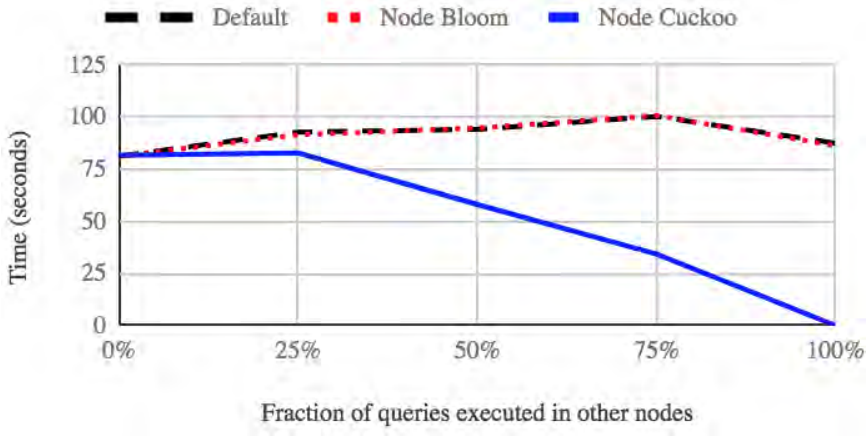


Figure 5.14: Lookup performance in AWS environment after data deletion, by varying fraction of queries retrieving rows from remote node. All the queries result in deleted data. Node Cuckoo filter improves lookup performance for up to 100%, depending on the fraction of queries executed in other nodes.

5.3.2.3 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 the remote node. We maintain that all of the queries that are executed in the connected node or the remote node results in some rows. Figure 5.16 shows the result of the experiment with queries returning few rows. The experiment

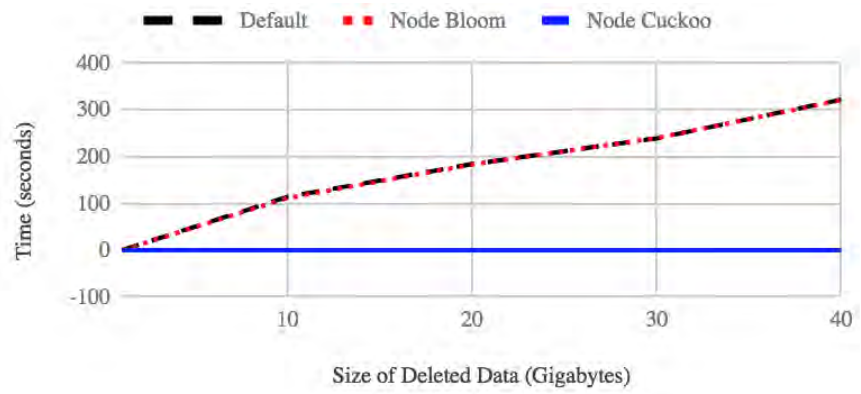


Figure 5.15: Lookup performance in AWS environment after data deletion, by varying deleted data size. All the queries are relayed to remote node for execution. Node Cuckoo filter improves lookup performance for up to 100%.

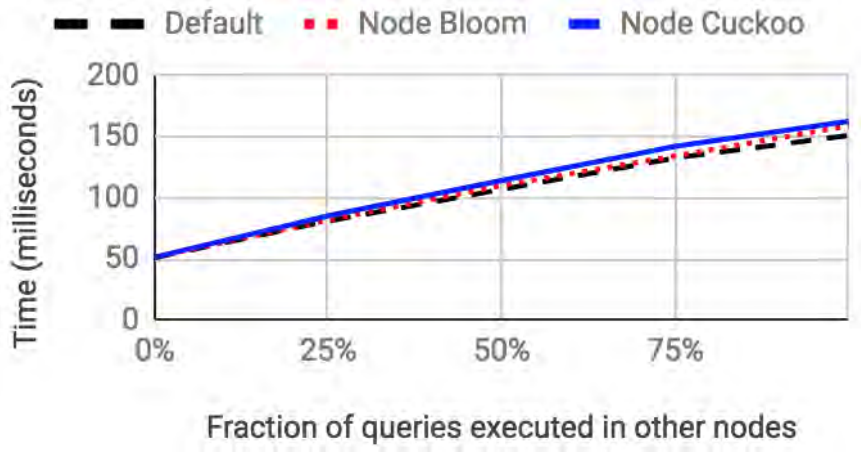


Figure 5.16: Lookup performance in simulated environment by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns positive results. Each query returns around 200 rows on average. The Node Filter implementations cause query execution time to decrease from 1 to 10 milliseconds. Each value is the average of 10 runs.

result shows that the overhead of looking up the node filter causes a very slight increase in query execution time. In case of our experiment, the query execution time increased from 1 to 10 milliseconds for the varying of fraction from 0% to 100%. The overhead is negligible considering the fact that actual query processing time exceeds seconds in Big Data and the millisecond overhead will go unnoticed. To prove this, we ran the same experiment with queries that return a large amount of data. Figures 5.17 and 5.18 show the result of the experiment that the performance is very similar for all implementations.

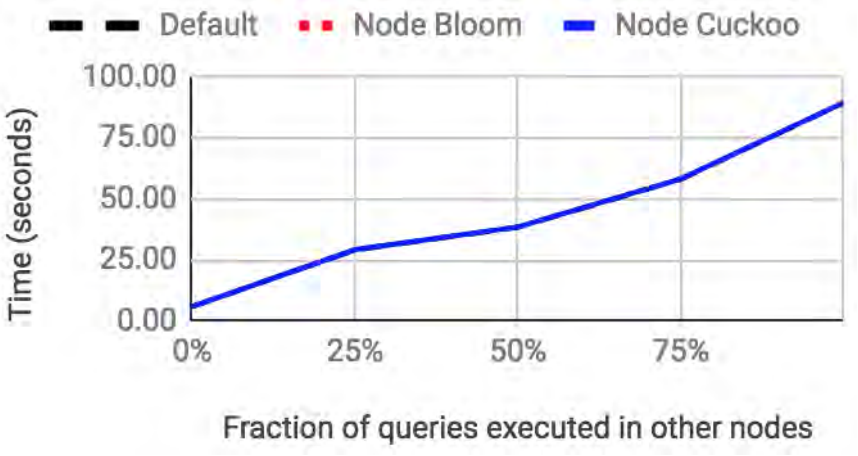


Figure 5.17: Lookup performance in simulated environment by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns positive results. Each query returns around 15K rows on average. The performances of all the implementations are very similar and hence overlap. Each value is the average of 10 runs.

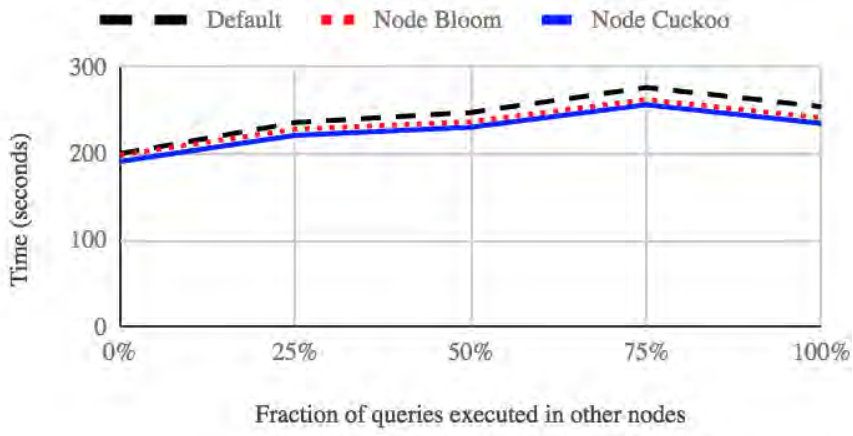


Figure 5.18: Lookup performance in AWS environment by varying fraction of queries executed in remote node, while all queries executed locally or remotely returns positive results. Each query returns around 5 million rows on average. The performances of all the implementations are very similar.

5.3.2.4 Insertion performance

Figure 5.19 shows the performance of data insertion (1M rows) in simulated environment, whereas Figure 5.20 shows the performance of data insertion (136M rows) in AWS environment. In both cases the results are almost the same for all the implementations. Therefore, our proposed scheme does not degrade performance in case of insertion queries.

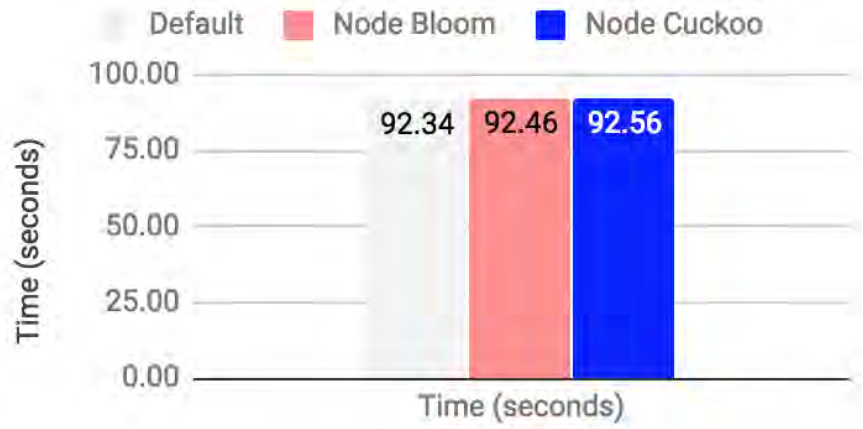


Figure 5.19: Insertion performance in simulated environment. Performance of all the implementations is very similar. Each value is the average of 10 runs.

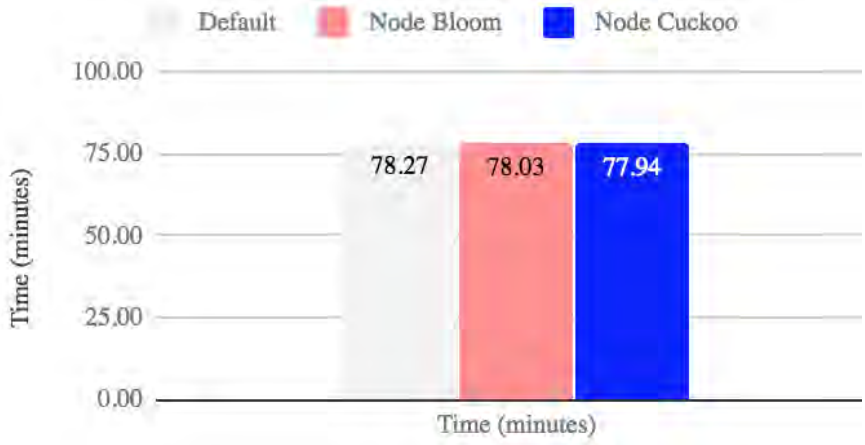


Figure 5.20: Insertion performance in AWS environment. Performance of all the implementations is very similar.

5.3.2.5 Effect of false positive rate on filter size and performance

From Figures 5.21 and 5.22 we can see that the less false positive probability we want to achieve, the more filter size becomes. Although the size of Cuckoo Filter is supposed to be smaller than Bloom Filter in case of false positive probability of less than 3%, and not larger in considerable amount for other cases, in our experiment we found the Cuckoo Filter size to be extremely high due to the implementation of the Cuckoo Filter library that we chose. It is possible to bring down the size to the expected amount by choosing a different Cuckoo Filter library implementation.

We also ran a lookup performance after deletion experiment for varying false positive prob-

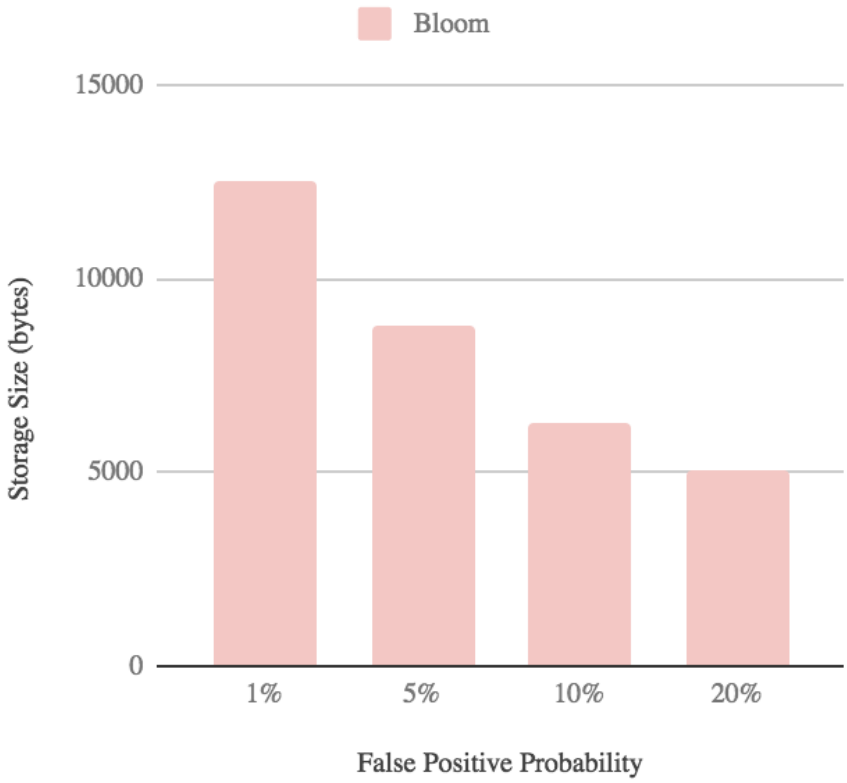


Figure 5.21: Node Bloom filter size for varying false positive probability (for 10K keys in AWS environment). Filter size increases for decrease in false positive proability.

ability and found that the performance of each filter type to be almost the same for any false positive probability (Figure 5.23). Although in practice false positive cases should decrease query execution performance, in our case we could not choose a set of experiment keys that would result in false positive occurrence, and hence we found the performances to be very similar. In future we would like to devise an experimental keyset using which we can perform this experiment and get result similar to real life applications.

5.4 Summary

In this chapter, we have proposed a scheme that improves query performance significantly in distributed Big Data systems by placing a probabilistic filter inside each node and looking it up before relaying queries to remote nodes in clusters only if data exists in those nodes. The

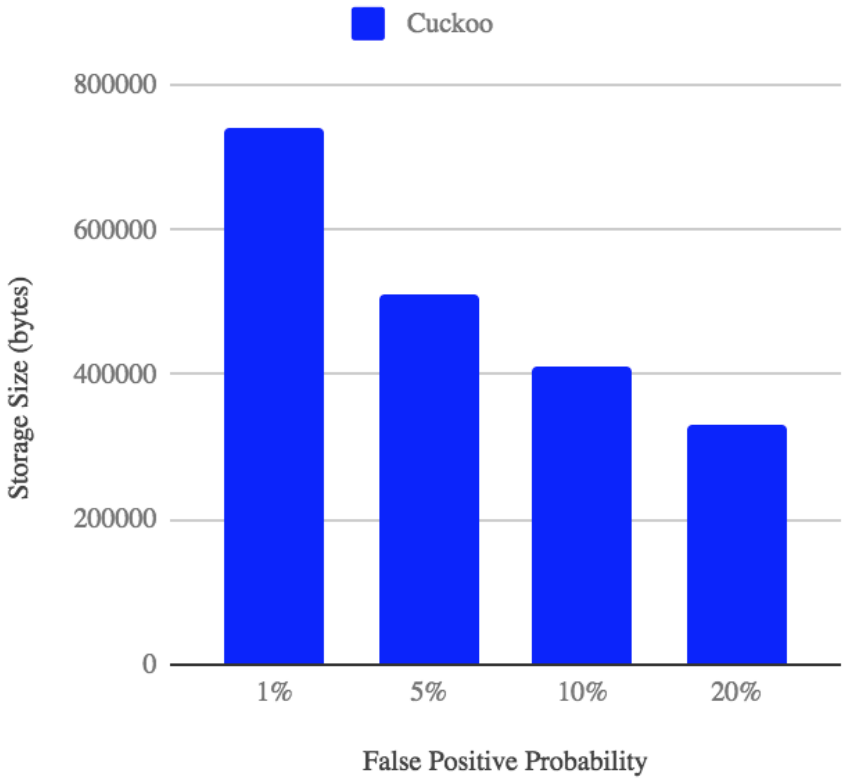


Figure 5.22: Node Cuckoo filter size for varying false positive probability (for 10K keys in AWS environment). Filter size increases for decrease in false positive proability.

scheme has been evaluated with a popular Big Data database (Cassandra) and it has been shown to improve performance of lookup queries for up to 100% in cases where data do not exist in remote nodes or have been deleted there.

The next chapter will conclude the thesis with some future plans.

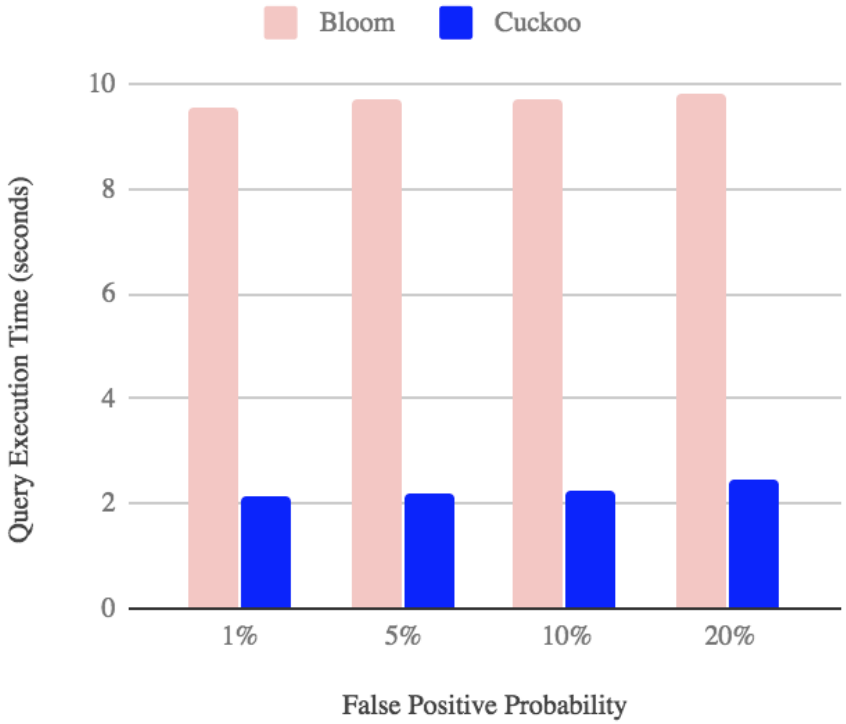


Figure 5.23: Lookup after deletion performance for varying false positive probability (for 10K keys in AWS environment). Performance of each filter type is very similar for all false positive rates as no two key resulted in the same bucket after hashing.

Chapter 6

Conclusion

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. However, none of the techniques address the issue of performance improvement by having pre-existing knowledge of partition data existence in each node of a Big Data cluster. Again, none of the techniques address another issue of performance degradation of lookup queries after data deletion in an Eventually Consistent Big Data system.

This research proposes two schemes that improve query performance significantly. One scheme proposes replacing Bloom Filter with a better probabilistic data structure called Cuckoo Filter that supports deletion of elements from within it. This leads to query performance improvement in Big Data systems that employs Eventual Consistency model, in the case of executing lookup queries after data deletion. The other scheme proposes placing a probabilistic filter inside each node and looking it up before relaying queries to remote nodes in clusters only if data exists in those nodes.

Both the schemes have been evaluated with a popular Big Data system (Cassandra) and each scheme has been shown to improve performance of lookup queries for up to 100%.

6.1 Future Work

In the future, we plan to extend our work achieving the followings:

- Exploring and evaluating other probabilistic data structures that may improve query performance further.
- Exploring 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.

Bibliography

[1] P. Venkatachar. (September 2014) Boost Oracle data warehouse performance using SanDisk solid state drives (SSDs). [Online]. Available: https://www.sandisk.com/content/dam/sandisk-main/en_us/assets/resources/enterprise/white-papers/boost-oracle-data-warehouse-performance-using-sandisk-ssds.pdf [Accessed: Aug. 15, 2018].

[2] L. P. Perea and V. Inozemtsev. From MySQL to Phoenix. [Online]. Available: https://visual-meta.com/tech-corner/from-mysql-to-phoenix.html [Accessed: Aug. 15, 2018].

[3] M. Winand. What every developer should know about SQL performance. [Online]. Available: https://use-the-index-luke.com/sql/dml/insert [Accessed: Aug. 15, 2018].

[4] DataStax Cassandra Documentation. How Cassandra reads and writes data - How is data written? [Online]. Available: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlHowDataWritten.html [Accessed: Aug. 15, 2018].

[5] DataStax Cassandra Documentation. How Cassandra reads and writes data - How is data read? [Online]. Available: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlAboutReads.html [Accessed: Aug. 15, 2018].

[6] B. Dupras. Probabilistic filters by example. [Online]. Available: https://bdupras.github.io/filter-tutorial/ [Accessed: Aug. 15, 2018].

[7] DataStax Cassandra Manual. Load balancing. [Online]. Available: https://docs.datastax.com/en/developer/java-driver/3.5/manual/load_balancing/ [Accessed: Aug. 15, 2018].

[8] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically better than Bloom," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, ser. CoNEXT '14. New York, NY, USA: ACM, 2014, pp. 75–88.

[9] P. Vagata and K. Wilfong, "Scaling the Facebook data warehouse to 300PB," April 2014. [Online]. Available: https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb [Accessed: Aug. 15, 2018].

[10] J. Frazier. (July 2015) Why data deletion makes sense (and dollars). [Online]. Available: http://www.ftijournal.com/article/why-data-deletion-makes-sense-and-dollars [Accessed: Aug. 15, 2018].

[11] Cisco visual networking index: forecast and methodology. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html [Accessed: Aug. 15, 2018].

[12] D. Anikin. (September 2017) Choosing between an in-memory and a traditional DBMS. [Online]. Available: https://dzone.com/articles/when-and-why-i-use-an-in-memory-database-or-a-trad [Accessed: Aug. 15, 2018].

[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[14] G. Fowler, L. C. Noll, K.-P. Vo, D. E. E. 3rd, and T. Hansen, "The FNV Non-Cryptographic Hash Algorithm," Internet Engineering Task Force, Internet-Draft draft-eastlake-fnv-15, Jun. 2018, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-15

[15] L. Chi and X. Zhu, "Hashing techniques: A survey and taxonomy," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 11:1–11:36, Apr. 2017.

[16] A. Goel and P. Gupta, "Small subset queries and Bloom filters using ternary associative memories, with applications," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[17] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of Bloom filters for distributed systems," *IEEE Communications Surveys Tutorials*, vol. 14, no. 1, pp. 131–155, First 2012.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems*, vol. 26, no. 2, pp. 4:1–4:26, Jun. 2008.

[19] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[20] T. Lee, K. Kim, and H. J. Kim, "Join processing using Bloom filter in MapReduce," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, ser. RACS '12. New York, NY, USA: ACM, 2012, pp. 100–105.

[21] M. Bhushan, S. Banerjea, and S. K. Yadav, "Bloom filter based optimization on HBase with MapReduce," in *Data Mining and Intelligent Computing, IEEE*, Sept 2014, pp. 1–5.

[22] R. Jain, M. Rawat, and S. Jain, "Data optimization techniques using Bloom filter in Big Data," *International Journal of Computer Applications*, vol. 142, no. 3, pp. 23–27, 2016.

[23] R. Pagh and F. F. Rodler, "Cuckoo hashing," *J. Algorithms*, vol. 51, no. 2, pp. 122–144, May 2004.

[24] Y. Sun, Y. Hua, S. Jiang, Q. Li, S. Cao, and P. Zuo, "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, ser. USENIX ATC '17. Berkeley, CA, USA: USENIX Association, 2017, pp. 553–565.

[25] A. D. Mauro, M. Greco, and M. Grimaldi, "A formal definition of Big Data based on its essential features," *Library Review*, vol. 65, no. 3, pp. 122–135, 2016.

[26] "Apache Cassandra on the Wikipedia," https://en.wikipedia.org/wiki/Apache_Cassandra. [Online]. Available: https://en.wikipedia.org/wiki/Apache_Cassandra [Accessed: Aug. 15, 2018].

[27] J. Casares, *Multi-datacenter Replication in Cassandra*, November 2012. [Online]. Available: https://www.datastax.com/dev/blog/multi-datacenter-replication [Accessed: Aug 15, 2018].

[28] "Cassandra Source Code on Github," https://github.com/apache/cassandra. [Online]. Available: https://github.com/apache/cassandra [Accessed: Aug. 15, 2018].

[29] "DB-Engines Ranking of Wide Column Stores," https://db-engines.com/en/ranking/wide+column+store. [Online]. Available: https://db-engines.com/en/ranking/wide+column+store [Accessed: Aug. 15, 2018].

[30] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 515–529, Sep. 2010.

[31] S. Richter, J.-A. Quiané-Ruiz, S. Schuh, and J. Dittrich, "Towards zero-overhead static and Adaptive Indexing in Hadoop," *The VLDB Journal*, vol. 23, no. 3, pp. 469–494, Jun. 2014.

[32] F. M. Schuhknecht, J. Dittrich, and L. Linden, "Adaptive Adaptive Indexing," in *ICDE*, 2018.

[33] R. Nehme and N. Bruno, "Automated partitioning design in Parallel Database Systems," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11. New York, NY, USA: ACM, 2011, pp. 1137–1148.

[34] T. J. Ameloot, G. Geck, B. Ketsman, F. Neven, and T. Schwentick, "Data partitioning for single-round multi-join evaluation in massively parallel systems," *SIGMOD Rec.*, vol. 45, no. 1, pp. 33–40, Jun. 2016.

[35] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden, "AdaptDB: Adaptive partitioning for distributed joins," *Proc. VLDB Endow.*, vol. 10, no. 5, pp. 589–600, Jan. 2017.

[36] J.-G. Lee, J. Han, X. Li, and H. Gonzalez, "TraClass: Trajectory classification using hierarchical region-based and trajectory-based clustering," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 1081–1094, Aug. 2008.

[37] E. Zamanian, C. Binnig, and A. Salama, "Locality-aware partitioning in parallel database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 17–30.

[38] S. Chen, "Cheetah: A high performance, custom data warehouse on top of MapReduce," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1459–1468, Sep. 2010.

[39] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *PVLDB*, vol. 4, no. 11, pp. 1111–1122, 2011.

[40] E. Jahani, M. J. Cafarella, and C. R, "Automatic optimization for MapReduce programs," *PVLDB*, vol. 4, no. 6, pp. 385–396, 2011.

[41] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[42] D. Guo, Y. Liu, X. Li, and P. Yang, "False negative problem of Counting Bloom Filter," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 5, pp. 651–664, May 2010.

[43] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for Counting Bloom Filters," in *Proceedings of the 14th Conference on Annual*

European Symposium - Volume 14, ser. ESA'06. London, UK, UK: Springer-Verlag, 2006, pp. 684–695.

[44] B. Vöcking, "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, Jul. 2003.

[45] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. Magalhaes, "The Deletable Bloom Filter: A new member of the Bloom family," *IEEE Communications Letters*, vol. 14, no. 6, pp. 557–559, June 2010.

[46] S. Cohen and Y. Matias, "Spectral Bloom filters," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 241–252.

[47] A. Kirsch, M. Mitzenmacher, and U. Wieder, "More robust hashing: Cuckoo hashing with a stash," *SIAM Journal on Computing*, vol. 39, no. 4, pp. 1543–1561, Dec. 2009.

[48] M. Mitzenmacher, S. Pontarelli, and P. Reviriego, "Adaptive Cuckoo filters," in *Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments*, 2018, pp. 36–47.

[49] H. Chen, L. Liao, H. Jin, and J. Wu, "The Dynamic Cuckoo Filter," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct 2017, pp. 1–10.

[50] T. Xiang, X. Li, F. Chen, Y. Yang, and S. Zhang, "Achieving verifiable, dynamic and efficient auditing for outsourced database in cloud," *Journal of Parallel and Distributed Computing*, vol. 112, pp. 97 – 107, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731517302800

[51] Z. Xie, W. Ding, H. Wang, Y. Xiao, and Z. Liu, "D-Ary Cuckoo Filter: A space efficient data structure for set membership lookup," in *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, Dec 2017, pp. 190–197.

[52] M. Kwon, V. Shankar, and P. Reviriego, "Position-aware Cuckoo Filters," in *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ser. ANCS '18. New York, NY, USA: ACM, 2018, pp. 151–153.

[53] M. Kwon, P. Reviriego, and S. Pontarelli, "A length-aware Cuckoo filter for faster IP lookup," in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, April 2016, pp. 1071–1072.

[54] M. Kwon, S. Vajpayee, P. Vijayaragavan, A. Dhuliya, and J. Marshall, "Use of Cuckoo filters with FD.Io VPP for software IPv6 routing lookup," in *Proceedings of the SIGCOMM Posters and Demos*. ACM, 2017, pp. 127–129.

[55] D. Eppstein, M. T. Goodrich, M. Mitzenmacher, and M. R. Torres, "2-3 Cuckoo Filters for faster triangle listing and set intersection," in *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, ser. PODS '17. New York, NY, USA: ACM, 2017, pp. 247–260.

[56] J. Cui, J. Zhang, H. Zhong, and Y. Xu, "SPACF: A secure privacy-preserving authentication scheme for VANET with Cuckoo filter," *IEEE Transactions on Vehicular Technology*, vol. 66, no. 11, pp. 10 283–10 295, Nov 2017.

[57] V. V. Mahale, N. P. Pareek, and V. U. Uttarwar, "Alleviation of DDoS attack using advance technique," in *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, Feb 2017, pp. 172–176.

[58] M. Al-Hisnawi and M. Ahmadi, "Deep packet inspection using Cuckoo filter," in *2017 Annual Conference on New Trends in Information Communications Technology Applications (NTICT)*, March 2017, pp. 197–202.

[59] Q. Xue and M. C. Chuah, "Cuckoo-filter based privacy-aware search over encrypted cloud data," in *2015 11th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, Dec 2015, pp. 60–68.

[60] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SlimDB: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2037–2048, Sep. 2017.

[61] S. I. M. Mosharraf, "Cassandra Source Code Forked on Github for Experimentation," https://github.com/sharafat/cassandra. [Online]. Available: https://github.com/sharafat/cassandra [Accessed: Aug. 15, 2018].

[62] M. Gunlogson, "CuckooFilter4J," https://github.com/MGunlogson/CuckooFilter4J. [Online]. Available: https://github.com/MGunlogson/CuckooFilter4J [Accessed: Aug. 15, 2018].

[63] "Data Expo '09," http://stat-computing.org/dataexpo/2009/the-data.html. [Online]. Available: http://stat-computing.org/dataexpo/2009/the-data.html [Accessed: Aug. 15, 2018].

[64] "Amazon Customer Reviews Dataset," https://registry.opendata.aws/amazon-reviews. [Online]. Available: https://registry.opendata.aws/amazon-reviews [Accessed: Aug. 15, 2018].

[65] S. I. M. Mosharraf, "Cassandra Experiments Executing Application," https://github.com/sharafat/cuckoo-filter-performance-analyzer. [Online]. Available: https://github.com/sharafat/cuckoo-filter-performance-analyzer [Accessed: Aug. 15, 2018].

[66] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, ser. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220.

[67] The Couchbase Team, "Couchbase," https://www.couchbase.com, 2010. [Online]. Available: https://www.couchbase.com [Accessed: Aug. 15, 2018].

[68] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project Voldemort," in *Proceedings of the 10th USENIX Con-*

ference on File and Storage Technologies, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 18–18.