CHAPTER 1

INTRODUCTION

A lot of data is being generated from a large variety of sources. The ability to store these large volumes of data has been increased from time to time. Since the volume of data is very large, the time required to process and perform any action on the stored data increased and hence more techniques were developed to improve the performance thereby reducing the processing time required to completely process the large volumes of data. Set-Membership problem is no exception and many of the probabilistic algorithms are proposed and developed to solve this problem in much lesser time (almost constant time).

1.1 PURPOSE

The purpose of this project is to improve the performance of the set-membership operations on Redis (in-memory database) by implementing Cuckoo-Filter algorithm. Currently Redis 3.2 has a Bloom-Filter algorithm implemented in an unstable branch for solving set-membership problem. But it is observed and proved that the Cuckoo-Filter algorithm provides much better performance than Bloom-Filter algorithm and hence this is an effort to utilize this improvement in performance in Redis by replacing Bloom-Filter with Cuckoo-Filter.

1.2 SCOPE

Project Goals and Deliverables:

a) Implement Cuckoo-Filter algorithm in Redis in C.
b) Determine the performance difference between Redis+Bloom and Redis+Cuckoo implementations.

Features:

a) Ability to ADD elements to the set on the fly without the need to re-create the bit vector again.
b) Ability to DELETE elements from the set on the fly without the need to re-create the bit vector again.
c) Improve the element LOOK-UP performance.
d) To make the set highly space efficient i.e. to make the bit vector almost completely filled before signaling that the vector is completely filled.

1.3 DEFINITIONS, ACRONYMS AND ABBREVIATIONS

In-Memory Databases (IMDB):

Databases are the storage repository system which provides the ability to manage data easily. Usually these databases store the data on the secondary storage areas (hard-disks), the persistent storage areas where the data is stored permanently even when there is no electrical energy supplied. Some examples of databases which store the data completely on secondary storage locations are MySQL, Oracle DB, Mongo DB, Cassandra and more. But these kinds of databases are not suitable for applications which require access to the data very frequently within a short span of time i.e. those applications where response-time is critical cannot use these traditional databases. Hence there was a requirement to develop the databases which provide faster access to the data stored in them. These types of databases are usually store the data in the primary memory (RAM). Hence these databases are known as In-Memory Databases.

In-Memory databases are the database systems which store the data in-memory thereby providing faster access to the data stored in them. Some examples of the in-memory databases are IBM DB2 database, Apache Geode database, Scuba (database developed by Facebook), Redis and many more.

Redis:

Redis is the In-Memory Database that stores the data in the form of key-value pairs developed and maintained by the Redis Labs. Redis stands for Remote Dictionary Server. Redis is also a NoSQL based database which means that the stored data does not have any structure and hence does not involve any structured query language for Redis. Instead, Redis uses the client-server based architecture for its operation. Any number of clients can connect to a single server and request for particular operation. Even though the data is stored in-memory in Redis, the stored data is temporarily backed-up on to the secondary storage to provide high data availability to the applications. The main advantages of using Redis are it provides High Scalability, High Availability and High Performance. Some of the applications which use Redis database are Twitter, GitHub, StackOverflow, Flickr, Slack, Instagram and others.

Set-Membership Problem:

The set-membership problem is the simplest problem, to determine if a particular element belongs to a set or not. But the complexity of the problem increases with the increase in the size of the set i.e. if the number of elements in the set are more, it takes much longer time to check for element existence by using the deterministic algorithms. Linear search is one such deterministic algorithm used to check for set-membership of an element. But the deterministic algorithms are not suitable for applications where the response time is critical and hence probabilistic algorithms were developed. Some of the probabilistic algorithms that are developed to solve set-membership problem are Bloom-Filter and Cuckoo-Filter algorithms.

Probabilistic Algorithms:

Probabilistic algorithms are the non-deterministic algorithms. Probabilistic algorithms are more suitable than the deterministic algorithms in situations where the time complexity is more important. The results obtained contain some percentage of false positives. If the probability of false positives is much smaller (i.e. 1 out of 10000) then the probabilistic algorithms are suitable than deterministic algorithms.

False Positives:

In the context of set membership problem, false positive is defined as the element which is wrongly identified to be present in the set i.e. the element is said to be present in the set even though it is really not belonging to the set.

True Positives:

In the context of set-membership problem, true positive is defined as the element which is correctly predicted to be present in the set i.e. the element is predicted to be present in the set and the element actually belongs to the set.

False Negatives:

In the context of set-membership problem, false negative is defined as the element which is wrongly predicted to be absent in the set i.e. the element is predicted to be absent in the set but the element actually belongs to set.

True Negatives:

In the context of set-membership problem, true negative is defined as the element that is correctly predicted not to be belonging to the set i.e. the element is predicted to be absent in the set and the element is actually not belonging to the set.

Recall Rate:

In the context of set-membership problem, recall rate is defined as the ratio of the number of elements that are predicted correctly to be belonging to the set to the total number of actual elements that belong to the set.

Hash Functions:

Hash functions are the functions which map the some arbitrarily sized data to the fixed size. The arbitrary sized data are called as keys and the fixed sized data are called as values. Given a key, the hash function returns the value corresponding to the given key. Some of the hash functions that are available today are SHA-1, MD5, CRC-32 etc.

Fingerprints:

Fingerprints are the small data values which uniquely identify a large data. Theoretically fingerprint for any given large data is unique, but practically it is very complex, difficult and impossible to generate unique fingerprints, multiple keys can have same fingerprint. Some of the Hash functions can themselves be used as fingerprint algorithms. Some of the algorithms used for generating fingerprints are Pearson's hashing, Rabin's fingerprint algorithm etc..

1.4 LITERATURE SURVEY

Bloom-filter is a probabilistic algorithm that was proposed by Burton Howard Bloom in 1970, for solving the set-membership problem. The Bloom-filter checks for the membership of the element in set in constant-time. This algorithm is applicable in situations where large amount of memory would be required for storing the data and error-free hashing methods are allowed or utilized for testing the existence of the element in the set.

Bloom-Filter provides some percentage of false positive results, but does not provide any false negative results, which means that the recall rate of Bloom-Filter is 100% i.e. for any given query to the Bloom-Filter, it returns either "possibly present in the set" or

"definitely not present in the set". Some of the areas which use Bloom-Filter are Akamai Web Servers, Google Big-Table, Apache Hbase, Apache Cassandra, Google Chrome and many more. Here is how the basic version of Bloom-Filter works.

Basic Bloom-Filter working:-

1) Initially the number of hash functions ('k') required for the given size of the bit-vector ('m' bits) is determined using the following formulae :-

   n – Given total number of elements in the filter.

   p – Given the required probability of false positives.

   m = ceil((n * log(P)) / log (1.0 / (pow (2.0 , log (2.0)))))

   k = round (log (2.0) * m / n)

   and all the bit array entries are set to value '0' indicating that the set is initially empty.

2) To add an element to the set, the element to be added is fed to each of the 'k' hash functions to obtain the 'K' index positions to the bit-array, then each entries corresponding to these 'K' index positions of the bit array are set to '1' indicating that the element is added to set.

3) To check for membership of the element in the set, the element is fed to each of the 'k' hash functions to obtain the 'K' index positions and the corresponding bit array entries are checked if they are set to '1'. If all the corresponding bit array entries are set to 1 then it is highly probable that the element may belong to the set, If at-least even one corresponding entry in the bit array is set to '0', then definitely that element is not present in the set.

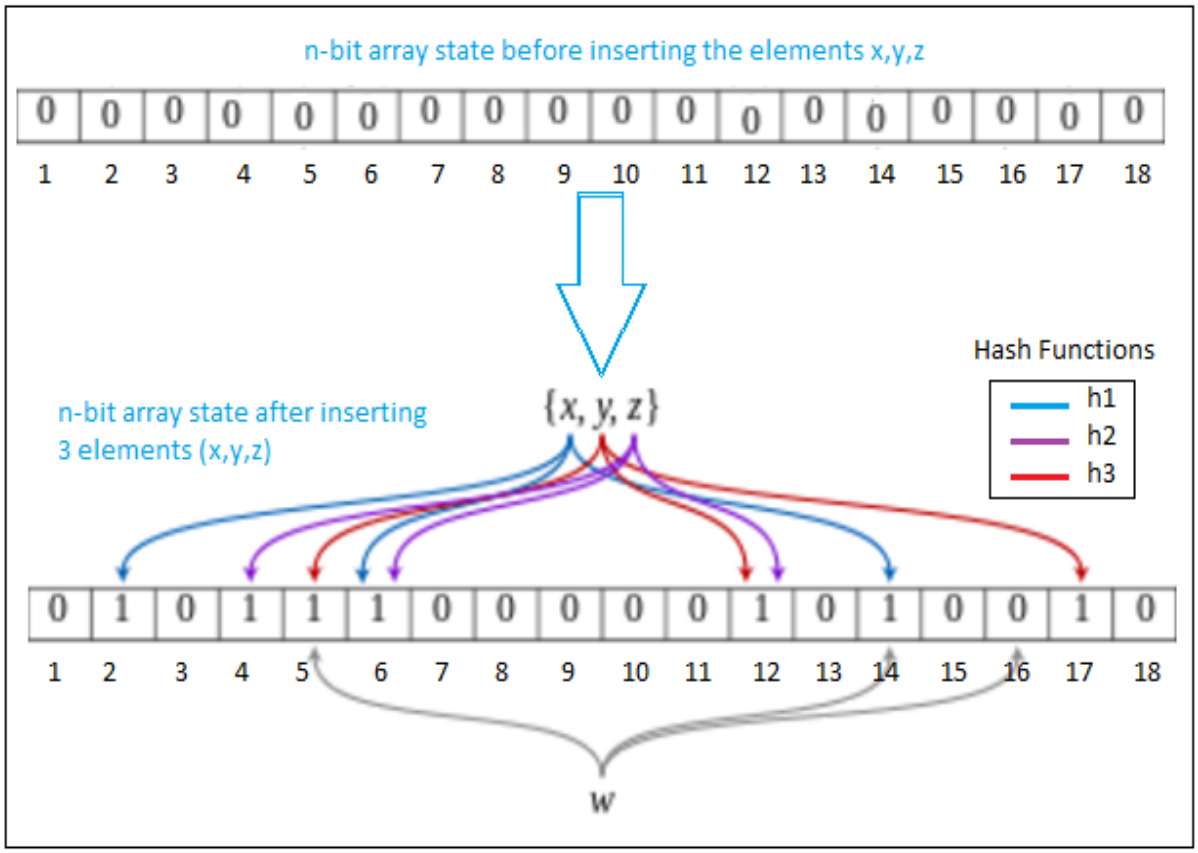

Figure 1 Working of Bloom-Filter

But it is clearly visible that the Bloom-Filter doesn't allow the deletion operation i.e. Bloom-Filter does not work correctly if elements are deleted from the set. Many variations of Bloom-Filter have been proposed.

The following provides an overview on the variations of Bloom-Filter.

a) Counting Bloom Filters.

The Counting filters support deletion of elements from the set without recreating the bit-array again. Here the bit-array entries instead of using 1 bit per item, uses n-bits per item. While inserting an item, the entry in the bit array is incremented by 1 each time and the look-up operation checks if the bit-array entry is non-zero. The deletion operation decrements the value by 1.

b) Layered Filters.

The layered bloom filters uses multiple filter layers. Each layers are individual basic bloom filters. These filters allow keeping track of the number of times a particular element is inserted or added into the set. The element set-membership

operations on these filters returns the deepest or the inner most layer within which the element is found.

c) Blocked Filters.

The blocked bloom filters contain array of the small basic bloom filters, similar to the layered bloom filters. These filters do not support deletion operation on any element of the set. These filters provide better spatial locality on the look-up operations. Each of the small filters fit in one CPU cache line and every item is stored only in any one of the small filters determined by the hash partitioning approach.

d) Quotient Bloom Filters.

The quotient bloom filters provide support for the deletion operations on the set without the need to recreate the bit vectors. These filters store the fingerprints of the given element instead of storing the binary digits ‘0’ or ‘1’. But these filters require an extra storage/memory than the blocked filters and layered filters.

Other variations of bloom-filters exist other than once mentioned in the above paragraph. The time complexity of the basic bloom filter is O(1). The recall rate of the bloom-filter is 100%. The false positive rate is around 2.5% if 8 bits are used for inserting/storing an element in the set. The following table describes the relationship between the false positive rate and the number of bits used for storing/inserting an element in the set. It is clearly visible from the following table that as the number of bits per element increases, the false positive rate decreases.

Table 1 False Positive Rate vs Bits Required per item

False Positive Rate	Bits required per item
50%	1.44
10%	4.79
2%	8.14
1%	9.58
0.1%	14.38
0.01%	19.17

The following diagram depicts the variation of the false positive rate of bloom filter with respect to the bits required per item.

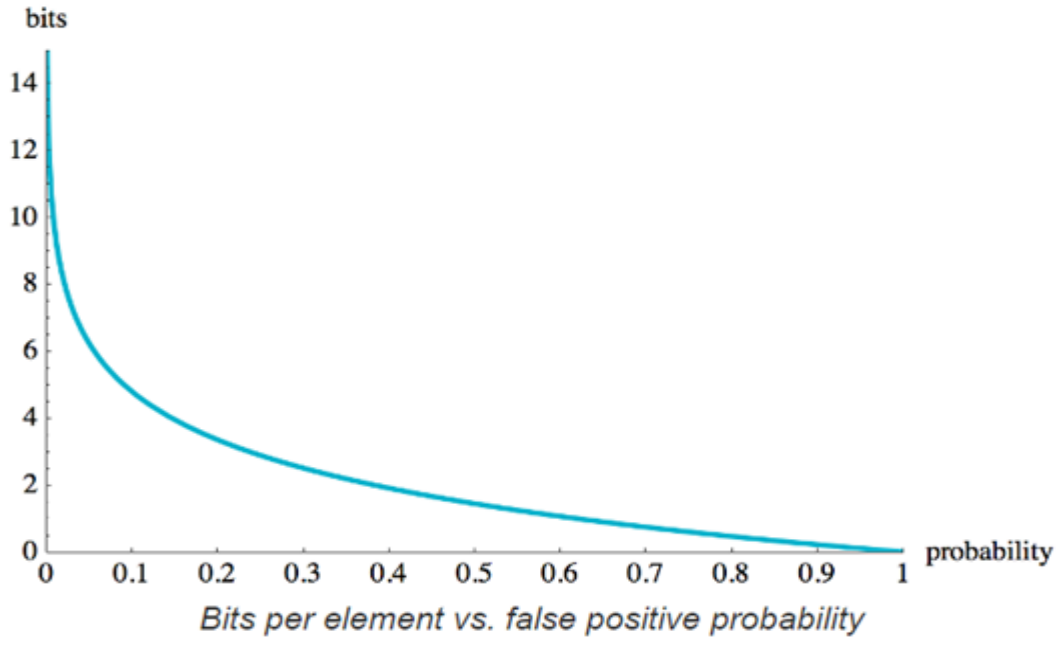

Figure 2 Bloom filter: Bits per Item vs False positive rate

Cuckoo Filter: By understanding how the bloom filter works, it is clear that the main disadvantage of using bloom filter is that it does not support the deletion operations on the elements of the set. Even though many of the variations of bloom filter were proposed,

suggested and implemented, there exist either the space trade-off or performance trade-off. Cuckoo filter is also a probabilistic algorithm that solves the set-membership problem. Cuckoo-filter is far better than the bloom-filter in the sense that the cuckoo filter allows the deletion operations on the elements of the set i.e. there's no need to recreate the bit vector even on deleting an element from the set. The elements can be deleted and added dynamically from the set in case of cuckoo filter implementation. Cuckoo filters are suitable for applications which store many elements and which require very low false positive rates. Compared to bloom filters cuckoo filters provide lower space overhead.

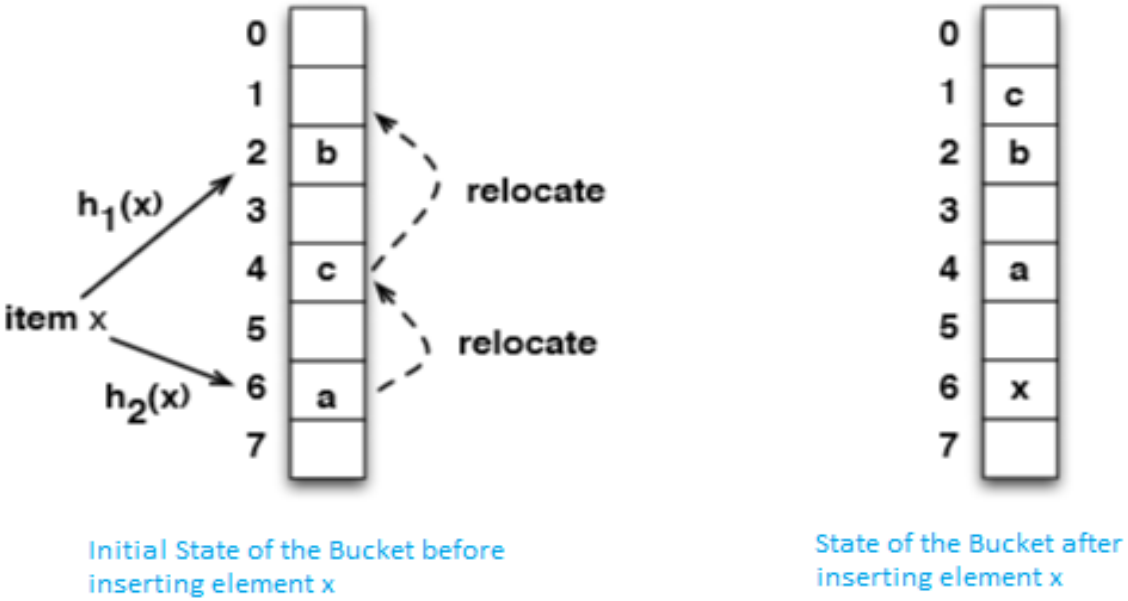

Figure 3 Overview working of Cuckoo-Hashing technique

Cuckoo filter uses Cuckoo-Hashing method for increasing the space occupancy of the elements in the set. A basic cuckoo hashing uses two hash functions 'h1(x)' and 'h2(x)' and uses single array of bucket 'a' as the bit-vector as shown in the above figure. For inserting an element say 'e', the element 'e' is given as input to both of the hash functions each of which gives the indexes 'i1' and 'i2' to the array of buckets. The element is inserted into any of these two entries a[i1] or a[i2]. If none of the entries of the buckets are free, then any one of the entry is selected and the element already present at this location is kicked out to a different entry of the array.

The following diagram depicts the performance difference between Cuckoo filter and Bloom filters.

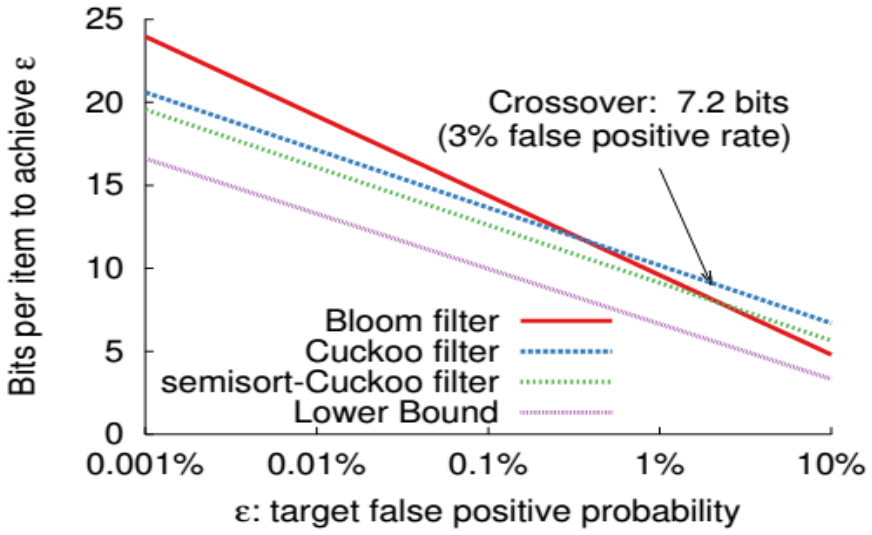

Figure 4 Cuckoo vs Bloom filters

1.5 EXISTING SYSTEM

Currently Redis uses the Bloom-filter algorithm to check for the element membership of the set.

The following illustrates the working of Redis with Bloom filter implementation for set-membership operation.

1) The Redis Client connects to the Redis Server requesting for a particular operation to be completed. There can be ‘n’ number of clients which can connect to the same Redis Server.

2) The redis-client requests the server to check for the existence of an element in a particular set.

3) The redis-server uses the bloom-filter technique to check for element membership in the set and sends response to the redis-client.

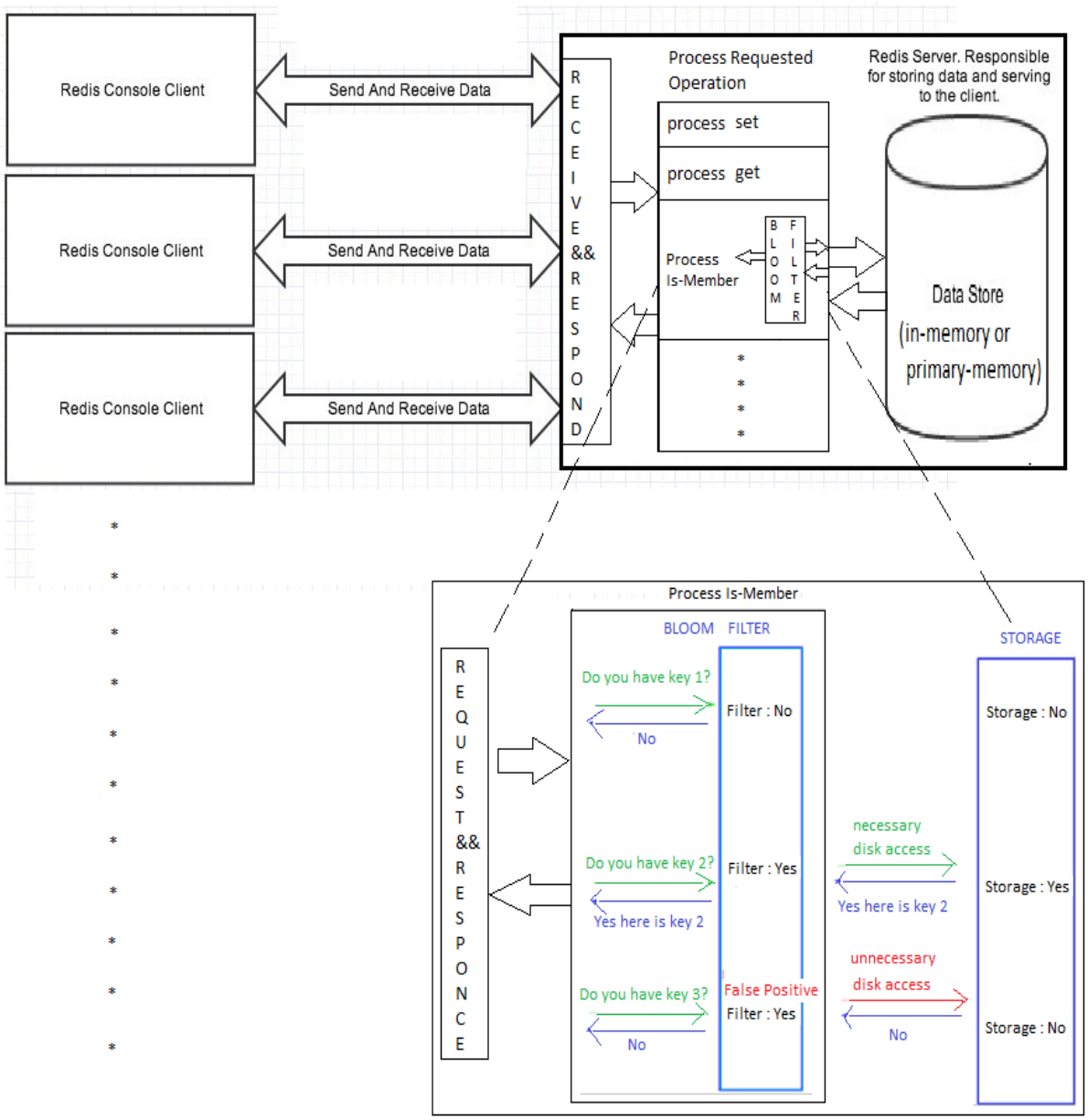

Figure 5 Actual working of Redis with Bloom filter

1.6 PROPOSED SYSTEM

The following illustrates the proposed solution of how Redis can use Cuckoo filter for its set-membership operations.

1) The Redis clients connect to the Redis server requesting for particular operation.
2) After the connection between client and server is successful, the client issues the checkIsMember() operation to the server whenever required.

3) When the server receives the checksIsMember() request from the client, it processes the request using the Cuckoo-Filter algorithm which produces the result in constant time O(1).
4) Once the result is determined, the server issues the response containing the result to the corresponding client that had requested for checkIsMember operation.
5) There can be 'n' number of clients which can request the server for a particular operation.

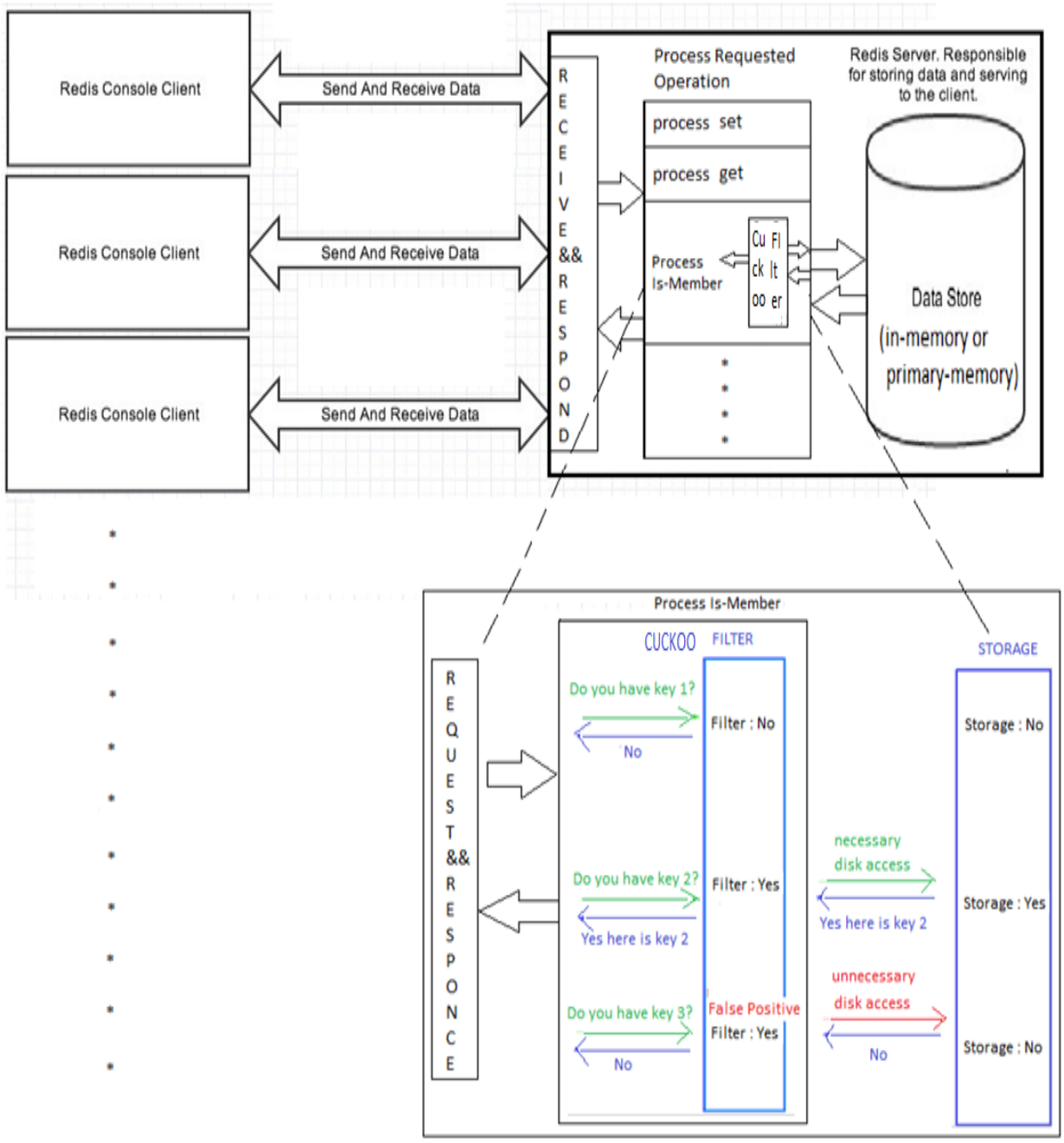

Figure 6 Proposed Working of Redis + Cuckoo Filter

1.7 PROBLEM STATEMENT

The aim of the project is to enhance the performance of Set-Membership operations on Redis (in-memory) database by replacing Bloom-filter implementation with the Cuckoo-filter implementation.