ABSTRACT

Nowadays, there exist hundreds of millions of data online. These data are either stored in web pages, or databases of companies, such as Facebook, Flickr, etc. It is challenging to quickly find similar data from these huge repositories. These data have different properties. We can use hashing algorithms to improve the searching efficiency. Locality sensitive hashing (LSH) is a similarity searching method. LSH works on the principle that if there are two points in feature space closer to each other, they are very likely to have same hash. A Bloom Filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set. In this project we will implement locality sensitive bloom filter on Raspberry Pi which will be an effective method for data searching.

TABLE OF CONTENTS

CONTENTS	PAGE NO.
ACKNOWLEDGEMENT	i
ABSTRACT	ii
LIST OF TABLES	iv
LIST OF FIGURES	v
1. INTRODUCTION	1
2. LITERATURE SURVEY	3
2.1 Tree Structure	3
2.2 Vector Approximation File	3
2.3 Approximation Tree	4
2.4 Hashing	4
3. BLOOM FILTER	6
3.1 Application and Examples	9
4. HASHING METHOD	10
4.1 Locality Sensitive Hashing	10
4.1.1 Application of LSH	11
4.2 Checksum	12
4.3 SHA-1	12
4.4 MD5	13
5. METHODOLOGY	14
5.1 Algorithm	14
5.2 Language and Libraries	15
6. RESULTS AND DISCUSSION	16
CONCLUSION	17
FUTURE WORK AND SCOPE	18
FUNDING	19
OTHER ACTIVITIES	19
REFERENCE	20

LIST OF TABLES

NO.	TITLE	PAGE NO.
6.1	Result when computed with 10,000 numbers	16
6.2	Result when computed with 100,000 numbers	16
6.3	Result when computed with 10,00,000 numbers	16

LIST OF FIGURES

NO.	TITLE	PAGE NO.
1.1	Tree-based machine learning algorithm	1
3.1	Representation of Bloom Filter	6
3.2	Digital Representation of Bloom Filter	
3.3	An Example Of A Bloom Filter Representing The Set {x,y,z}	
4.1	Comparison between General Hashing and LSH	8
5.1	Block Diagram	

CHAPTER 1

INTRODUCTION

Driven by the tremendous growth of information in a massive dataset, there is an increasing need for an efficient similarity searching method that can locate desired information rapidly with low cost. An ideal similarity searching scheme should work well in a high-dimensional database. Specifically, it should locate nearly all the similar records of a query with a short query time and small memory requirement. In addition, it should also deal with data insertion and deletion.

Some of the similarity searching methods rely on tree structures, such as R-tree, SS-tree and SRtree. These data structures partition the data objects based on their similarity. Tree-based machine learning algorithm are popular in partitioning and clustering objects, see Figure 1.1. However, for high-dimensional objects, partitioning is not efficient.

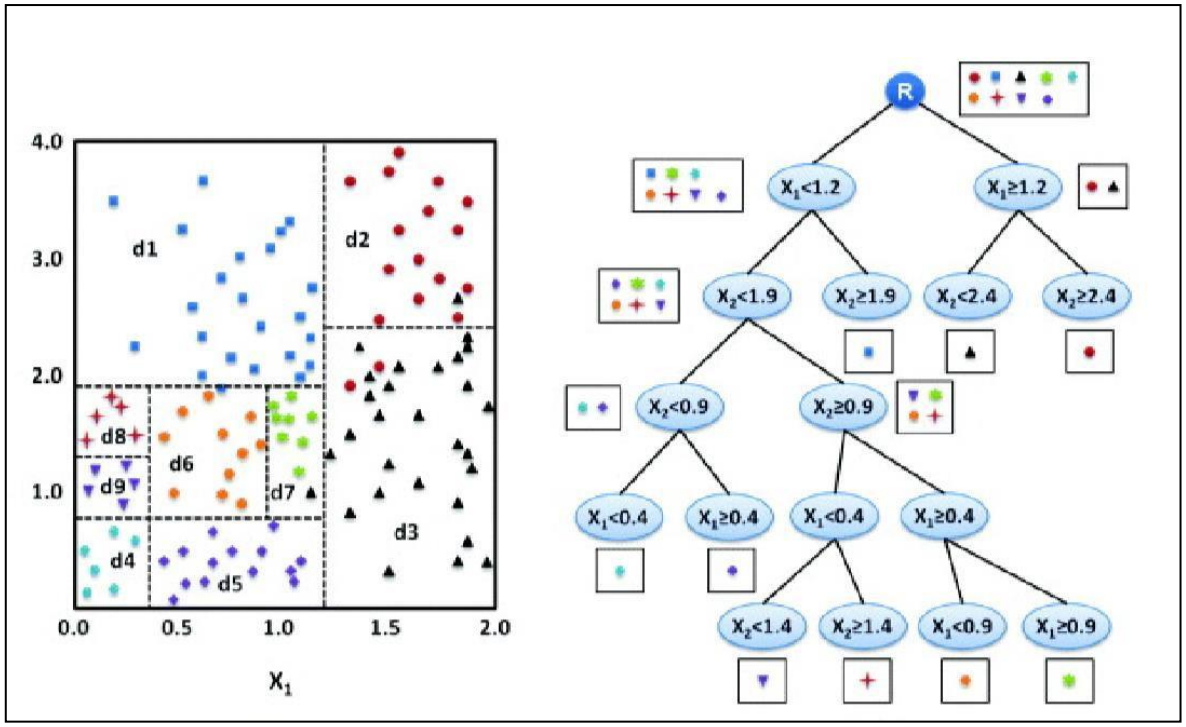


Figure 1.1 Tree-based machine learning algorithm

We investigated LSH to approach this problem, because it excels at finding near neighbors in sub-linear time. LSH is better for finding near or nearest neighbors in high dimensions than other algorithms, because it scales well.

A bloom filter is a space efficient probabilistic data structure which can quickly test for set membership. However, since a standard bloom filter will only test for presence, it has limitations. This project shows the feasibility of Locality Sensitive Bloom Filters (LSBF) for single-dimensional data and points to further, theoretical, algorithms for distance sensitive hashes that can be used efficiently for multidimensional data. A Bloom filter is a data structure designed to tell you, rapidly and memory-efficiently, whether an element is present in a set.

Today billions of spam data are spreading day by day in our world. Till now we don't have a efficient method to search for spam data from this huge amount of data. Bloom filter is an efficient data structure which can be used for storing and querying of data. In this project we enhanced the capability of bloom filter by adding an efficient hashing method called locality sensitive hashing so that data searching can be done much faster. This proposed methodology can be used in social networks and cyber security in order to prevent the spreading of Spam Data.

CHAPTER 2

LITERATURE SURVEY

The similarity searching problem is closely related to the nearest neighbor search problem, which has been studied by many researchers. Various indexing data structures have been proposed for nearest neighbor searching.

2.1 Tree structures

Some of the similarity searching methods rely on tree structures, such as R-tree, SS-tree and SRtree. These data structures partition the data objects based on their similarity. Therefore, during a query, only a part of the data records has to be compared with the query record, which is more efficient than the linear search that compares a query with every data record in the database. Though these data structures can support nearest neighbor searching, they are not efficient in a large and high-dimensional database (i.e., the dimensionality is more than 20). The M-tree was proposed to organize and search large datasets from a generic metric space, i.e., where object proximity is only defined by a distance function satisfying the positivity, symmetry, and triangle inequality postulates. The M-tree partitions objects on the basis of their relative distances measured by a specific distance function, and stores these objects into nodes that correspond to constrained regions of the metric space. All data objects are stored in the leaf nodes of M-tree. The non-leaf nodes contain “routing objects” which describe the objects contained in the branches. For each routing object, there is a so-called covering radius of all its enclosing objects, and the distances to each child node are pre-computed. When a range querying is completed, sub-trees are pruned if the distance between the query object and the routing object is larger than the routing object’s covering radius plus the query radius. Because a lot of the distances are pre-computed, the query speed is dramatically increased. The main problem is the overlap between different routing objects in the same level.

2.2 Vector approximation file

Another kind of similarity searching method is the vector approximation file (VA-file) , which can reduce the amount of data that must be read during similarity searches. It divides the data space into grids and creates an approximation for each data object that fall into a grid. When searching for the near neighbors, the VA-file sequentially scans the file containing these approximations, which is smaller than the size of the original data file. This allows most of the

VA-file's disk accesses to be sequential, which are much less costly than random disk accesses. One drawback of this approach is that the VA-file requires a refinement step, where the original data file is accessed using random disk accesses.

2.3 Approximation tree

Approximation tree (A-tree) has better performance than VA-file and SR-tree for high dimensional data searching. The A-tree is an index structure for similarity search of high-dimensional data. A-tree stores virtual bounding rectangles (VBRs), which contain and approximate minimum bounding rectangles (MBR) and data objects, respectively. MBR is a bounding box to bind data object. iDistance partitions the data into different regions and defines a reference point for each partition. The data in each region is transformed into a single dimensional space based on their similarity with the reference point in the region. Finally, these points are indexed using a B+-tree structure and similarity search is performed in the one-dimensional space. As reported in, iDistance outperforms the M-tree and linear search.

2.4 Hashing

Hashing is a common approach to facilitate similarity search in high dimension databases, and spectral hashing is one state-of-the-art work for dataaware hashing. Spectral hashing applies the machine learning techniques to minimize the semantic loss of hashed data resulting from embedding. However, the drawback of Spectral hashing lies in its limited applicability. As spectral hashing relies on Euclidean distance to measure the similarity between two data records, and it requires that data points are from a Euclidean space and are uniformly distributed. Most recently, much research also has been conducted on locality-sensitive hashing. Dasgupta et al. proposed a new and simple method to speed up the widely-used Euclidean realization of LSH. At the heart of the method is a fast way to estimate the Euclidean distance between two d-dimensional vectors; this is achieved by the use of randomized Hadamard transforms in a non-linear setting. Traditionally, several LSH functions are concatenated to form a "static" compound hash function for building a hash table. Gan et al. proposed to use a base of m single LSH functions to construct "dynamic" compound hash functions, and defined a new LSH scheme called Collision Counting LSH (C2LSH). In C2LSH, if the number of LSH functions under which a data object o collides with a query object q is greater than a pre-specified collision threshold, then o can be regarded as a good candidate of c-approximate nearest neighbors of q. Slaney and Casey described an LSH technique that allows one to quickly find similar entries in large databases. This approach belongs to a novel

and interesting class of algorithms that are known as randomized algorithms, which do not guarantee an exact answer but instead provide a high probability guarantee of returning correct answer or one close to it. Recent work has also explored ways to embed high dimensional features or complex distance functions into a low-dimensional Hamming space where items can be efficiently searched. However, existing methods do not apply for high-dimensional kernelized data when the underlying feature embedding for the kernel is unknown. Kulis and Grauman showed how to generalize locality-sensitive hashing to accommodate arbitrary kernel functions, making it possible to preserve the algorithm's sub-linear time similarity search guarantees for a wide class of useful similarity functions. Semantic hashing seeks compact binary codes of data-points so that the Hamming distance between codewords correlates with semantic similarity. Weiss et al. showed that the problem of finding a best code for a given dataset is closely related to the problem of graph partitioning and can be shown to be NP hard. By relaxing the original problem, they obtained a spectral method whose solutions are simply a subset of thresholded eigenvectors of the graph Laplacian. Satuluri and Parthasarathy proposed BayesLSH, a principled Bayesian algorithm for performing candidate pruning and similarity estimation using LSH. They also presented a simpler variant, BayesLSH-Lite, which calculates similarities exactly. BayesLSH can quickly prune away a large majority of the false positive candidate pairs. The quality of BayesLSH's output can be easily tuned and does not require any manual setting of the number of hashes to use for similarity estimation.

CHAPTER 3

BLOOM FILTER

It was developed by Burton Howard Bloom in 1970. Bloom filters are called filters because they are often used as a cheap first pass to filter out segments of a dataset that do not match a query. The Bloom filter is a space-efficient probabilistic data structure that supports set membership queries. The data structure was conceived by Burton H. Bloom in 1970. The structure offers a compact probabilistic way to represent a set that can result in false positives (claiming an element to be part of the set when it was not inserted), but never in false negatives (reporting an inserted element to be absent from the set). This makes Bloom filters useful for many different kinds of tasks that involve lists and sets. The basic operations involve adding elements to the set and querying for element membership in the probabilistic set representation. The basic Bloom filter does not support the removal of elements; however, a number of extensions have been developed that also support removals. The accuracy of a Bloom filter depends on the size of the filter, the number of hash functions used in the filter, and the number of elements added to the set. The more elements are added to a Bloom filter, the higher the probability that the query operation reports false positives.

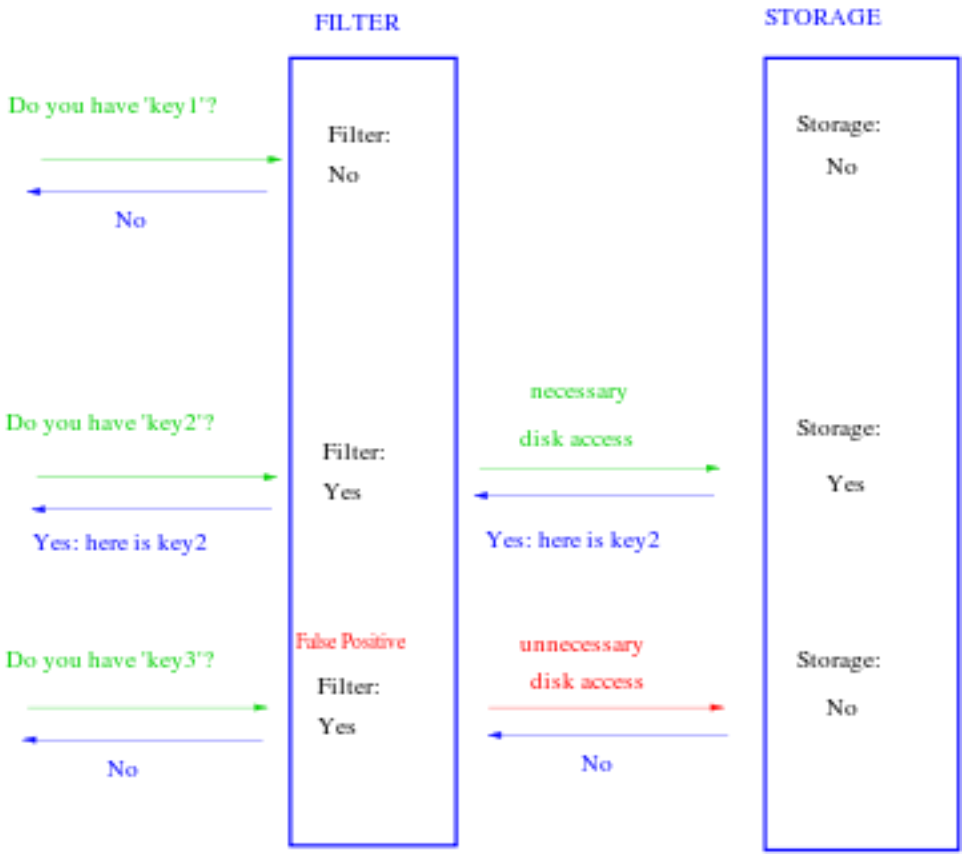


Figure 3.1 Representation of Bloom Filter

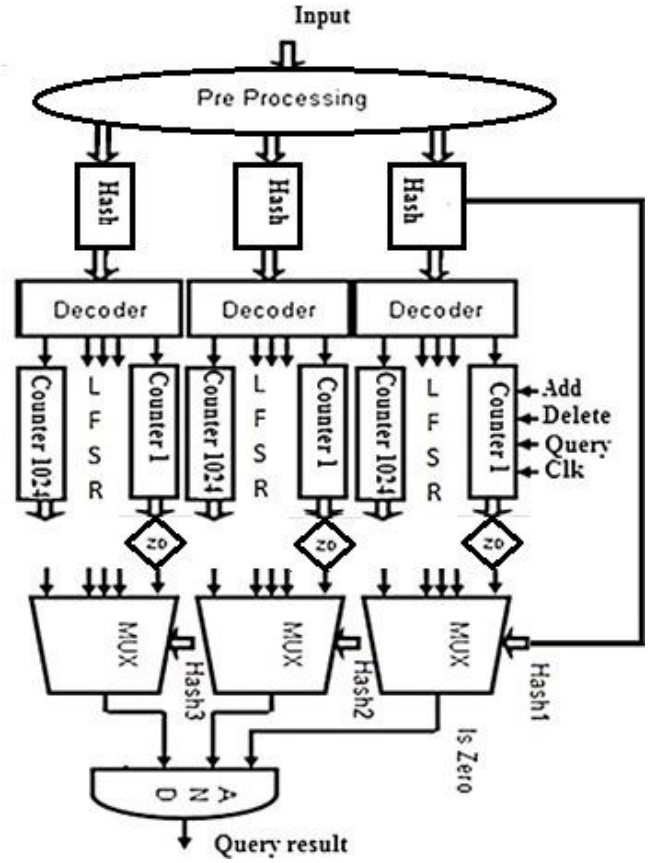


Figure 3.2 Digital representation of Bloom Filter

While Bloom Filters can say "definitely not in" with confidence, they will also say possibly infor some number of items. Depending on the application, this could be a huge downside or it could be relatively okay. If it is okay to introduce false positives every now and then, you should definitely consider using Bloom Filters for membership existence for set operations.

Also note that if you are decreasing the false positive rate arbitrarily, you would increase the number of hash functions which would add latency to both insertion and membership existence. One more thing in this section is that, if the hash functions are independent each other and distribute the input space pretty uniformly, then the theoretic false positive rate can be satisfied. Otherwise, the false positive rate of the bloom filter will be worse than the theoretic false positive rate as hash functions correlate each other and hash collisions would occur more often than desired.

Bloom Filter is a bit array of N bits, where N is the size of the bit array. It has another parameter which is the number of hash functions, k. These hash functions are used to set bits in the bit array. When inserting an element x into the filter, the bits in the k indices h1(x),h2(x), ..., hk(x) are set, where the bit positions are determined by the hash functions. Note that as we

increase the number of hash functions, the false positive rate of this probability goes to zero. However, it takes more time to insert and lookup as well as the bloom filter fills up more quickly.

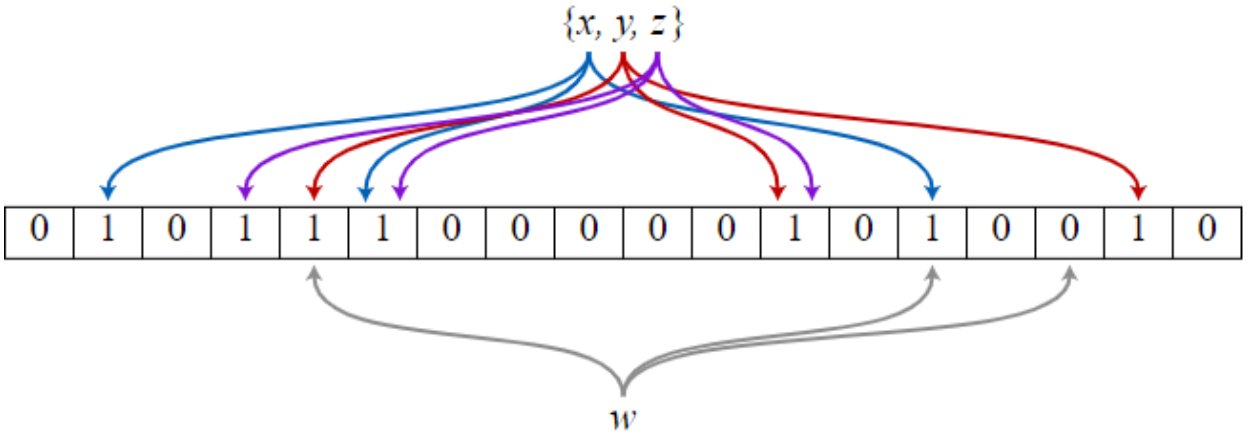


Figure 3.3 An example of a Bloom filter, representing the set {x, y, z}.

The colored arrows show the positions in the bit array that each set element is mapped to. The element w is not in the set {x, y, z}, because it hashes to one bit-array position containing 0. For this figure, m = 18 and k = 3.

For each element $x \in S$, the bit $h_i(x)$ is set to 1 for $1 \le i \le k$. To check if an item y is in S, we check whether all $h_i(y)$ are set to 1. If not, then clearly y is not a member of S. If all $h_i(y)$ are set to 1, then y is a member of S but with the possibility of some false positive results (FPR). Hence false positive means treating a nonmember as a member. The probability of a false positive result for an element can be calculated as follows,

The probability that a specific bit to be 0 is,

$$(1-1/m)^{kn} = e^{-kn/m.} \qquad (1)$$

Let $p = e^{-kn/m}$ then the probability of false positive or the probability that a specific bit to be 1 is,

$$(1-(1-1/m)^{kn})^{k} = (1-e^{-kn/m})^{k} = P \qquad (2)$$

This gives the mathematical representation of false positive probability ratio (FPR) and denoted as P. This FPR cannot be completely avoided but it can be relatively reduced by choosing less number input elements and larger arrays of bloom filter which is evident from the above equations and also by choosing appropriate hashing functions yielding further optimization to fix low FPR in prior.

3.1 APPLICATIONS & EXAMPLES

- Google BigTable, Apache HBase and Apache Cassandra use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.
- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).
- Medium uses Bloom filters to avoid recommending articles a user has previously read.
- Bitcoin uses Bloom filters to speed up wallet synchronization.
- The Squid Web Proxy Cache uses Bloom filters for cache digests.
- Spell checkers

CHAPTER 4

HASHING METHOD

4.1 LOCALITY SENSITIVE HASHING

In this section, we introduce LSH, LSH-based similarity searching method, and min-wise independent permutations. LSH is an algorithm used for solving the approximate and exact near neighbor search in high dimensional spaces. The main idea of the LSH is to use a special family of hash functions, called LSH functions, to hash points into buckets, such that the probability of collision is much higher for the objects which are close to each other in their high dimensional space than for those which are far apart. A collision occurs when two points are in the same bucket. Then, query points can identify their near neighbors by using the hashed query points to retrieve the elements stored in the same buckets. For a domain S of a set of points and distance measure the LSH is a hashing-based algorithm to identify approximate nearest neighbors. In the normal nearest neighbor problem, there are a bunch of points (let's refer to these as training set) in space and given a new point, objective is to identify the point in training set closest to the given point. Complexity of such process is linear [for those familiar with Big-O notation, O(N), where N is the size of training set]. An approximate nearest neighboring algorithm tries to reduce this complexity to sub-linear (less than linear but can be anything). Sub-linear complexity is achieved by reducing the number of comparisons needed to find similar items.

LSH works on the principle that if there are two points in feature space closer to each other, they are very likely to have same hash (reduced representation of data). LSH primarily differs from conventional hashing (aka cryptographic) in the sense that cryptographic hashing tries to avoid collisions but LSH aims to maximize collisions for similar points. In cryptographic hashing a minor perturbation to the input can alter the hash significantly but in LSH, slight distortions would be ignored so that the main content can be identified easily. The hash collisions make it possible for similar items to have a high probability of having the same hash value.

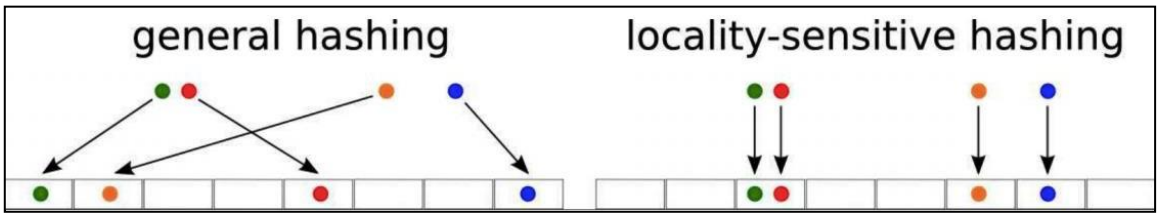


Figure 4.1 Comparison between General Hashing and LSH

4.1.1 Applications of LSH

Before jumping into understanding LSH, it is worth noting that the application areas are varied, including:

- Recommendation systems
- Near-duplicate detection
- Hierarchical clustering
- Genome-wide association study
- Image similarity identification (VisualRank)
- Audio similarity identification
- Digital video fingerprinting

Uber used LSH to detect platform abuses (fake accounts, payment fraud, etc.). A Shazam styled app or YouTube sized recommender system can be built using LSH.

4.2 CHECKSUM

A checksum is the outcome of running an algorithm, called a cryptographic hash function, on a piece of data, usually a single file. Comparing the checksum that you generate from your version of the file, with the one provided by the source of the file, helps ensure that your copy of the file is genuine and error free.

A checksum is also sometimes called a hash sum and less often a hash value, hash code, or simply a hash.

4.3 SHA-1

SHA-1 (short for Secure Hash Algorithm 1) is one of several cryptographic hash functions. SHA-1 is most often used to verify that a file has been unaltered. This is done by producing a checksum before the file has been transmitted, and then again once it reaches its destination. The transmitted file can be considered genuine only if both checksums are identical.

SHA-1 is only one of the four algorithms in the Secure Hash Algorithm (SHA) family. Most were developed by the US National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST).

SHA-0 has a 160-bit message digest (hash value) size and was the first version of this algorithm. SHA-0 hash values are 40 digits long. It was published under the name "SHA" in 1993 but wasn't used in many applications because it was quickly replaced with SHA-1 in 1995 due to a security flaw.

SHA-1 is the second iteration of this cryptographic hash function. SHA-1 also has a message digest of 160 bits and sought to increase security by fixing a weakness found in SHA-0. One real-world example where SHA-1 may be used is when you're entering your password into a website's login page. Though it happens in the background without your knowledge, it may be the method a website uses to securely verify that your password is authentic.

In this example, imagine you're trying to login to a website you often visit. Each time you request to log on, you're required to enter your username and password. If the website uses the SHA-1 cryptographic hash function, it means your password is turned into a checksum after you enter it in. That checksum is then compared with the checksum that's stored on the website that relates to your current password, whether you haven't changed your password since you signed up or if you just changed it moments ago. If the two match, you're granted access; if they don't, you're told the password is incorrect.

4.4 MD5

MD5 (technically called MD5 Message-Digest Algorithm) is a cryptographic hash function whose main purpose is to verify that a file has been unaltered.

Instead of confirming that two sets of data are identical by comparing the raw data, MD5 does this by producing a checksum on both sets and then comparing the checksums to verify that they're the same. MD5 hashes are 128-bits in length and are normally shown in their 32 digit hexadecimal value equivalent. This is true no matter how large or small the file or text may be.

Here's an example:

- Plain text: **This is a test.**
- Hex value: **120EA8A25E5D487BF68B5F7096440019**

When more text is added, the hash translates to a totally different value but with the same number of characters:

- Plain text: **This is a test to show how the length of the text does not matter.**
- Hex value: **6c16fcac44da359e1c3d81f19181735b**

CHAPTER 5

METHODOLOGY

Two hash functions were chosen to generate the hashes, SHA1 and MD5. MD5 is not considered cryptographically safe, but this doesn't matter in a Bloom filter, we just need a hash function that is uniform in its distribution and independent. MD5 is also a quick cryptographic hash so is of particular use to us in a Bloom filter. That being said, there is strong evidence that MurmurHash is a good hash to use in Bloom filters despite not being considered a cryptographic hash. We also take into account an iterative searching algorithm.

5.1 ALGORITHM

- Creating a data
- Storing the data in bloom filter.
- Retrieving the data from bloom filter using LSH and compute the time taken for retrieving the data.
- Repeat the step using linear searching algorithm and compute the time taken for retrieving the data.
- Using SHA1 And MD5 hashing retrieve the data from bloom filter and compute the time taken for retrieving the data.
- Compare the result and analyse.

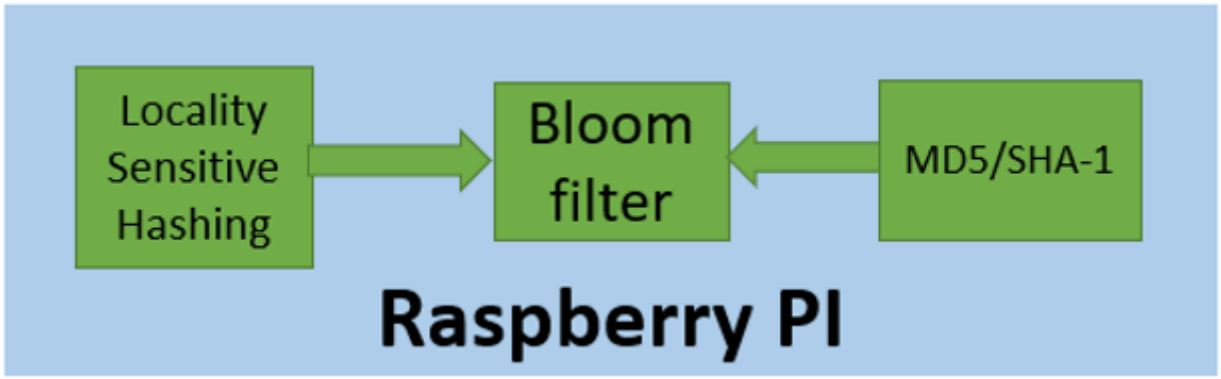


Figure 5.1 Block diagram

5.2 LANGUAGE AND LIBRARIES

The programming language Python was chosen for its ease of prototyping and vast libraries covering most required functionality. Python version 2.7 was used as most libraries support it and there is extensive documentation for it online.

The libraries *math* and *hashlib* are native to Python and imported for some basic mathematical functionality and to make use of some built-in hashing algorithms. The library "*bitarray*" is an efficient object type representing an array of booleans. It is implemented in C and abstracted to Python for use in Python programs. It offers much functionality but the small memory footprint is of most use to this project.

Two hash functions were chosen to generate the hashes, SHA1 and MD5. MD5 is not considered cryptographically safe, but this doesn't matter in a Bloom filter, we just need a hash function that is uniform in its distribution and independent. MD5 is also a quick cryptographic hash so is of particular use to us in a Bloom filter

CHAPTER 6
RESULTS AND DISCUSSION

Search Algorithm	Execution Time
Iterative method	38.068 µs
LSBF method	55.652 µs

Table 6.1 Result when computed with 10000 numbers.

Each method was run with a value number = 10000 and randomly generated numbers in the range 0 – 10000 tested as the query and 1000 values stored. It appears here that Iterative performs the best but when we increase n to 100,000 we can see something interesting

Search Algorithm	Execution Time
Iterative method	55.484 µs
LSBF method	15.388 µs

Table 6.2 Result when computed with 100,000 numbers.

Here, LSBF vastly outperforms an iterative approach, likely due to the fact that n has increased beyond the range of the numbers we're picking from. This shows how effective LSBF is when a positive result is likely. With the range of numbers randomly selected increased to 0 – 10,000,000 we see this:

Search Algorithm	Execution Time
Iterative method	7828.85866 µs
LSBF method	35.589 µs

Table 6.3 Result when computed with 10,00,000 numbers.

This final result shows that for a large range and large n, LSBF still outperforms an iterative search.

CONCLUSION

While comparing with iterative and our LSBF search algorithm, it was found that for a small amount of data the iterative search outperforms LSBF but in the case of high density of data we were able to spot that the LSBF method performs faster than iterative search method by a considerable margin. In this project we propose a method for efficient data searching which can be implemented for various social networks and cybersecurity. We know millions of spam messages are being spread day by day through social networks. Till now we don't have an efficient method to solve this spreading or identification of spam messages. But our proposed project Locality sensitive bloom filter can be used as a solution for this problem. Moreover, this system can be used as a firewall to block unwanted data. From the result we obtained we can identify that our proposed project takes less time for data searching and only take less processing power. Locality sensitive bloom filter can be used for searching Text files and multimedia files (Video and images).

FUTURE WORK AND SCOPE

Research in the field of Locality Sensitive Hashing is ongoing with promising space and time efficiency. Implementing a hashing algorithm would have vast applications in image recognition, databases, IP traceback and DNA sequencing.

FUNDING

Received a funding of rupees ₹.14,000/- provided by KSCSTE (KERALA STATE COUNCIL FOR SCIENCE, TECHNOLOGY AND ENVIRONMENT) under the scheme INNOVATE to promote innovations from Engineering students in the challenging areas of Science and Technology or socially relevant technologies in the Kerala State

OTHER ACTIVITIES

- Participated in KTU Techfest TEKON-2019 held at Government Engineering College Thrissur.
- Participated in SRISHTI-2019 National level technical project exhibition and competition held at Saintgits College of Engineering.

REFERENCE

1. https://www.slideshare.net/hamidfzm/bloom-filter-56450239
2. https://santhoshhari.github.io/Locality-Sensitive-Hashing/
3. https://towardsdatascience.com/fast-near-duplicate-image-search-using-locality-sensitive-hashing-d4c16058efcb
4. https://towardsdatascience.com/locality-sensitive-hashing-for-music-search-f2f1940ace23
5. https://towardsdatascience.com/fast-near-duplicate-image-search-using-locality-sensitive-hashing-d4c16058efcb
6. http://maciejczyzewski.me/2014/10/18/bloom-filters-fastand-simple.html
7. Bloom, Burton H. (1970), "Space/Time Trade-offs in Hash Coding with Allowable Errors", Communications of the ACM 13 (7): 422–426, doi:10.1145/362686.362692
8. https://en.wikipedia.org/wiki/Bloom_filter
9. http://billmill.org/bloomfilter-tutorial/
10. http://corte.si/posts/code/bloom-filter-rules-of-thumb/
11. J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. Bioinformatics 2001, 5(17), 419–428.
12. P. Ciaccia, M. Patella, and P. Zezula. M-trees: an efficient access method for similarity search in metric space. In Proc. of the 23rd International Conference on Very Large Data Bases, Athens, Greece, August 25-29, 1997.
13. A. Dasgupta, R. Kumar, and T. Sarlos. Fast locality-sensitive hashing. In Proc. of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD), San Diego, USA, August 21-24, 2011.
14. A. Z. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” Internet Mathematics, vol. 1, no. 4, pp. 485–509, 2003.
15. B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.
16. A. Ostlin and R. Pagh, “Uniform hashing in constant time and linear space,” in STOC ’03: Proc. Thirty-fifth Annual ACM Symposium on Theory of Computing. New York, NY, USA: ACM, 2003, pp. 622– 628.
17. C. Henke, C. Schmoll, and T. Zseby, “Empirical evaluation of hash functions for multipoint measurements,” SIGCOMM Computer Communications Review, vol. 38, no. 3, pp. 39–50, 2008.
18. A. Pagh, R. Pagh, and S. S. Rao, “An optimal Bloom filter replacement,” in

SODA ’05: Proc. Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 823–829

19. E. Porat, “An optimal Bloom filter replacement based on matrix solving,” in CSR ’09: Proc. Fourth International Computer Science Symposium in Russia on Computer Science - Theory and Applications. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 263–273

20. M. Mitzenmacher, “Compressed Bloom filters,” in PODC ’01: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing. New York, NY, USA: ACM, 2001, pp. 144– 150.

21. P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, and Y. Tang, “On the false-positive rate of Bloom filters,” Information Processing Letters, vol. 108, no. 4, pp. 210–213, 2008.

22. D. Guo, J. Wu, H. Chen, and X. Luo, “Theory and network applications of dynamic Bloom filters,” in Proc. 25th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), April 2006.

23. R. L. Rivest. (1992, Apr.) RFC1321 - the MD5 message-digest algorithm. [Online]. Available: http://www.faqs.org/rfcs/rfcl321.html

24. T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, T. Lehman, and B. Schott, "Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512," in Proc. of 5th International Conference on Information Security (ISC 2002), Sao Paulo, Brazil, Oct. 2002, pp. 75-89.