International Research Journal of Computer Science (IRJCS) ISSN: 2393-9842
Issue 06, Volume 4 (June 2017) www.irjcs.com

Duplicate File Analyzer using N-layer Hash and Hash Table

Siladitya Mukherjee
Assistant Professor, Department of Computer Science,
St. Xavier's College (Autonomous), Kolkata, India
siladitya.mukherjee@sxccal.edu

Pramod George Jose
M.Sc. Graduate, Department of Computer Science,
St. Xavier's College (Autonomous), Kolkata, India
pramod.the.programmer@gmail.com

Soumick Chatterjee
M.Sc. Graduate, Department of Computer Science,
St. Xavier's College (Autonomous), Kolkata, India
soumick@live.com

Priyanka Basak
M.Sc. Graduate, Department of Computer Science,
St. Xavier's College (Autonomous), Kolkata, India
mailpriyankabasak20@gmail.com

Manuscript History
Number: IRJCS/RS/Vol.04/Issue06/JNCS10083
Received: 15, May 2017
Final Correction: 29, May 2017
Final Accepted: 26, May 2017
Published: June 2017
Citation: Mukherjee, S.; Jose, P. G.; Chatterjee, S. & Basak, P. (2017), 'Duplicate File Analyzer using N-layer Hash and Hash Table', Master's thesis, St. Xavier's College (Autonomous), Kolkata, India.
Editor: Dr.A.Arul L.S, Chief Editor, IRJCS, AM Publications, India
Copyright: ©2017 This is an open access article distributed under the terms of the Creative Commons Attribution License, Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited

Abstract— with the advancement in data storage technology, the cost per gigabyte has reduced significantly, causing users to negligently store redundant files on their system. These may be created while taking manual backups or by improperly written programs. Often, files with the exact content have different file names; and files with different content may have the same name. Hence, devising an algorithm to identify redundant files based on their file name and/or size is not enough. In this paper, the authors have proposed a novel approach where the N-layer hash of all the files are individually calculated and stored in a hash table data structure. If an N-layer hash of a file matches with a hash that already exists in the hash table, that file is marked as a duplicate, which can be deleted or moved to a specific location as per the user's choice. The use of the hash table data structure helps achieve O(n) time complexity and the use of N-layer hashes improve the accuracy of identifying redundant files. This approach can be used for folder specific, drive specific or a system wide scan as required.
Index Terms— File Management, File Organization, Files Optimization, Files Structure, Secondary storage, Hash table representations, File comparison, File signature, Hash, N-layer hash

I. INTRODUCTION

1.1 Problem definition

The storage space available on secondary memory, especially hard disks, has grown almost exponentially ever since the first hard disk was invented by IBM in 1956 [1].

© 2014- 17, IRJCS- All Rights Reserved

Due to this, more and more people have become negligent about what they store on their storage devices and they often end up storing a lot of redundant files. This leads to unnecessary files being stored on storage media which may result in more chances of fragmentation as the number of files are relatively more, longer disk access rates, longer time required for file scanning programs like anti-malware software, etc. This also creates a major problem on Solid State Drives (SSDs) which are based on NAND-flash technology and hence much faster than traditional Hard Disk Drives (HDDs), but are relatively costlier. Therefore, it becomes important to save disk space as much as possible.

Another major concern is associated with regular backups. The importance of taking regular backups of important files is a well-known fact. The problem with this approach is that most of the time, files are backed up more than once. Apart from incremental backup solutions, a full back-up or a differential backup solution will copy all the files, or parts of files that have changed since the last full backup, even if the system already has identical copies of those files in a previous backup [2], thereby eating up precious storage space. This also applies to situations when a user takes a full backup of his files before performing a clean installation of his computer's operating system. He may already have most of the files on his external media, but the user copies all the files and then forgets about the redundant files. Manually going through all the files, identifying the redundant files and then deleting or moving them is a herculean task and is impractical.

1.2 Objective

Existing solutions try to identify identical files by comparing the meta-data of the files, like the size of the files, the name of the file itself or any such combination. This is not an efficient solution as there can be two files with the same name and size, but may have different content.

One approach to solve this problem is to perform a byte-by-byte comparison between each file and every other file in the system, but the time complexity of such an algorithm would be $O(n^2)$. To make things worse, such an algorithm would have to access the storage media a lot, thereby increasing the execution time. This problem is further worsened while comparing multimedia files which are generally much larger than text files. The objective is to design such a system that can uniquely identify files and mark them as redundant or non-redundant, in less than $O(n^2)$ time complexity while also using as less memory and processing power as possible.

II. PROPOSED SOLUTION

2.1 Hash functions and their utility

A hash function is a one-way mathematical function which transforms an input message of arbitrary length into a unique hash value of fixed length in such a way that it is infeasible to compute the input message, given the hash value. The resulting hash value is also called a message digest or checksum which serves as a "signature" of the input message. Even a small change made to the input file or message will yield a completely different output hash and this is known as the avalanche effect [3]. In essence, a particular message will generate a unique message digest; which can only be generated by that message or file [4]. Some examples of popular hash algorithms are- the family of Secure Hash Algorithm (SHA), the family of RIPEMD (RACE Integrity Primitives Evaluation Message Digest), Message Digest 5 (MD5), Tiger, Whirlpool, etc.

The characteristic that can undoubtedly distinguish files is the signature or the message digest of each file. By using a hash table as the data structure to store the file signatures (hashes), a time complexity of $O(n)$ can be achieved. These file signatures are of fixed size - 48 bytes long (32 bytes for SHA256 and 16 bytes for MD5) for SHA256-MD5 combination, irrespective of the input file size; which is much smaller than the size of the original file, and are stored in the primary memory which further reduces the execution time. For example, if we are comparing a movie file of, say 2 GB, we would just have to consider 48 bytes (for SHA256-MD5 combination) instead of 2 GB.

2.2 Hash tables

A hash table [5] is a data structure that implements an unordered associative array [6]. An associative array is an abstract data type which consists of a collection of key-value pairs. An associative array, to some extent, is similar to an indexed one-dimensional array available in most programming languages. The search time complexity of an indexed array is $O(1)$ only if the index of the desired value is known ahead of time. Otherwise, an appropriate search algorithm has to be used on the array.

The time complexity in such a case would depend upon the chosen search algorithm. This need for a search algorithm arises because there is no relation between an index and the value stored at that index. The insertion time complexity for an array always remains as $O(1)$. Hash tables use a hash function to map a key with its corresponding value. The hash function is applied on the key to obtain an index and the value is stored at that index in the hash table. In this way, the key used while inserting a value into the hash table can later be used to retrieve that value in $O(1)$ time.

© 2014- 17, IRJCS- All Rights Reserved

The key is analogous to the index in case of an indexed array. For example, consider a situation where customer names and their locations have to be stored and retrieved. A 2D array can be used, where each element of the first column would store a customer's name and the element of the second column and the same row would contain the customer's location.

Index	Name	Location
0	Arnold	New York
1	Robin	Glasgow
2	William	San Francisco
3	Stella	València
4	Sachin	Mumbai

Fig. 1. 2D array to store Customer Name & Location

This solution would allow insertions and deletions in O(1) time; but if we would need to search for the location of a particular customer, then first we would have to search through the first column and search for the desired name, say, using the simplest linear search algorithm. Once the customer name is matched, then the corresponding customer's location can be fetched. In this case, the worst-case time complexity would be O(n). On the other hand, a hash table can be used, where the customer's name can be treated as the key and the customer's location as the value. Now, we can use the customer's name as key to insert the customer's location into the hash table as value and during retrieval we would just need to provide the customer's name and the hash table would be able to fetch the customer's location in O(1) time complexity.

Key	Value
Arnold	New York
Robin	Glasgow
William	San Francisco
Stella	València
Sachin	Mumbai

Fig. 2. Hash Table to store Customer Name & Location

2.3 N-layer hashing for improved collision resistance

Hash functions map n-length messages into a fixed size hash. The set of all such n-length messages is much bigger than the set of possible fixed length hashes, and this mapping of a larger input set onto a smaller output set will essentially result in a collision as per the pigeonhole principle. Consider a hash function like SHA-256 that produces an output of 256 bits from an arbitrarily large input [7]. Since it must generate one of 2^{256} outputs for each member of a much larger set of inputs, some inputs will definitely hash to the same output hash. This is called collision.

The signature of a file should be unique, but, there can be situations where the hash of two different files result in the same hash. For example, in the well-known hash algorithm, MD5, there is a probability of collision in every 2^{64} hashes, which is similar to the birthday paradox. To overcome this problem, two or more layers of hash algorithms can be employed. In other words, the output hashes of different hash functions, applied on the same file, are concatenated together. This improves collision resistance as the probability of collisions occurring in all the hash functions is very low. If hash algorithms which use the same block size are used, then this solution can be further improved by reading a block from the input file only once and then sending it to the respective hash functions. Thus, multiple hashes of the same file can be computed in a single pass of the input file. This will not result in an increase in the time complexity of the algorithm and it would still remain as O(n). In this paper, the authors have used SHA256 and MD5 combination as the file signature and both use 64 byte blocks.

2.4 Brief description

The authors in this paper propose a solution to this problem by analyzing all the files in a computer system, or a part of it (like a folder), and classifies each file as redundant (if an exact copy of the file already exists) or non-redundant, by comparing the file signatures. The proposed solution in default mode makes use of a hashing algorithm to generate the file signature, viz. SHA-256, and uses a combination of SHA-256 and MD5 for improved accuracy, if specified by the user. Any other hash algorithm can be used instead of SHA-256 and MD5. A dictionary [8], [9] data structure in .NET is employed which is a strongly typed hash-table where the best case search and insertion time complexity is O(1). The signature of each file (SHA256 or SHA256-MD5 combination) is calculated which is used as the key and the path of that file is used as the value which makes up a key-value pair.

© 2014- 17, IRJCS- All Rights Reserved

The signature generated for each file is first looked up in the dictionary to ensure whether the file already exists, in which case, it is classified as a redundant copy and the path of the file is added to a list of redundant files. If the file does not exist, the signature and the path of the file is added to the dictionary as a key-value pair. At the end of the scanning process, from the list of redundant files, the user can choose the files to delete, the files to move and the files to keep as it is.

III. SOLUTION DESCRIPTION

3.1 Data flow diagram

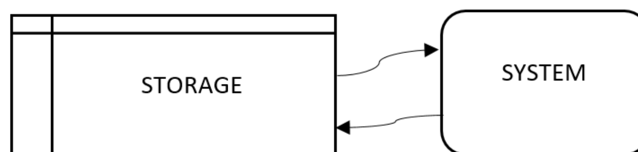

Fig. 3. Level 0 DFD

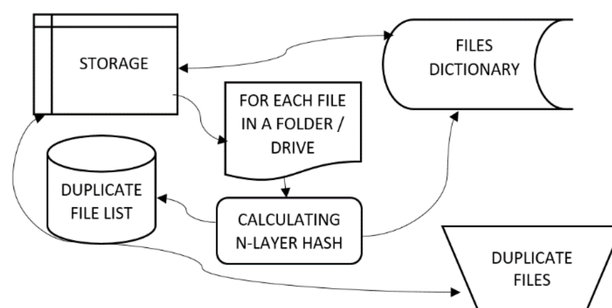

Fig. 4. Level 1 DFD

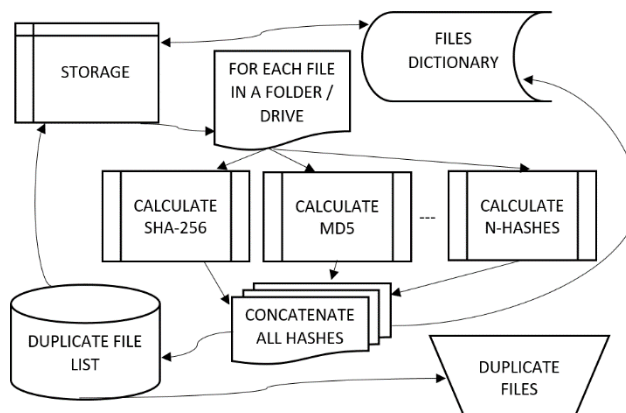

Fig. 5. Level 2 DFD

3.2 Algorithm

Algorithm DuplicateFileAnalyzer:

1. Select scan location i.e., a folder, an internal or removable drive or the entire computer.
2. Repeat until all files in the selected scan location, including all the sub-folders have been analyzed.
    a. Compute SHA-256 of the file.
    b. Compute MD5 of the file.
    c. Compute N - number of different hashes
    d. Concatenate all hashes.
    e. If the concatenated hash already exists as a key in the files dictionary, add the full path of the current file to the duplicate files list. Else, add a new key-value pair to the files dictionary, where key would be this concatenated hash and the value would be the full path of the current file.
3. For each file in the duplicate files list: -
    a. Select whether to delete or move the current file or to keep it as it is.
    b. Perform the selected operation.

© 2014- 17, IRJCS- All Rights Reserved

3.3 Flowchart

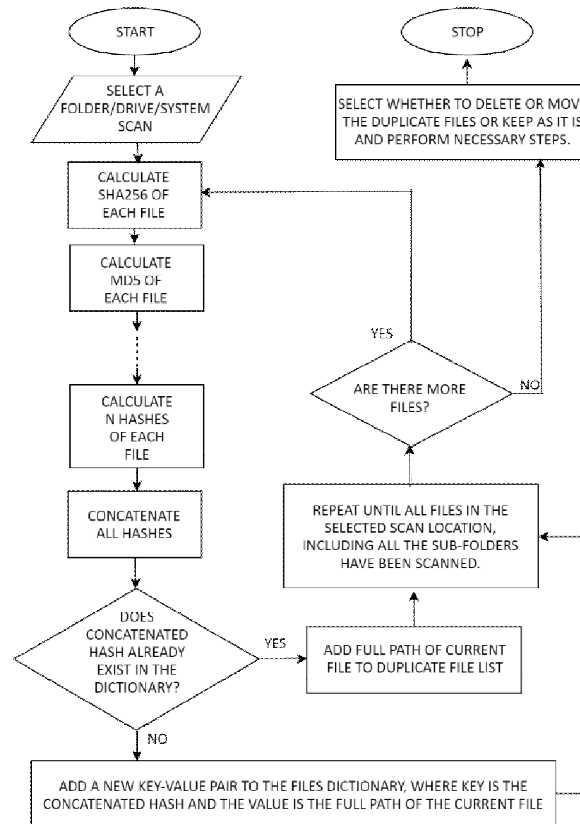

Fig. 6. Flowchart of DuplicateFileAnalyzer Algorithm

IV. RESULTS AND DISCUSSIONS

4.1 Features

- Recursive directory file comparison.
- Use of two hashes for reducing the chances of hash collisions.
- Use of two hashes of same block size, for computing both hashes simultaneously.
- Use of dictionary data structure for O(1) lookup.

4.2 Tests and results

Consider the following folder hierarchy: -

```
C:\tester>tree /F
Folder PATH listing
Volume serial number is A215-FC7A
C:.
│   MyTest.txt
│   MyTest2.txt
│
└───fol2
    │   MyTest.txt
    │   testing.mp3
    │
    └───fol3
            test.mp4
```

Fig. 7. File-Folder Structure which is used for Testing

In this structure, MyTest.txt, MyTest2.txt and fol2\MyTest.txt are files with unique contents. fol2\testing.mp3 and fol2\fol3\test.mp4 have the same content as MyTest.txt, but differ in their name and extension. After this file structure is processed by the demo application created using this proposed solution, it correctly identifies testing.mp3 and test.mp4 as duplicate files, even though the file names and extensions are different.

© 2014- 17, IRJCS- All Rights Reserved

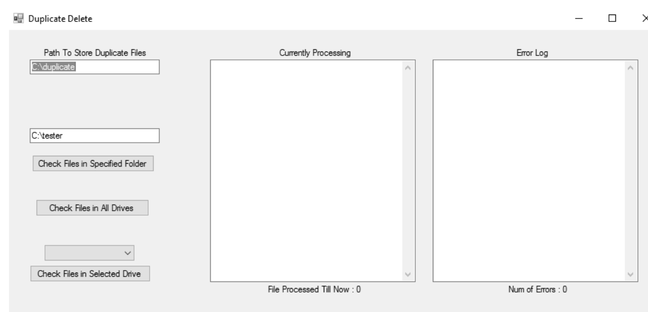

Fig. 8. Screenshot of demo application before starting the analysis

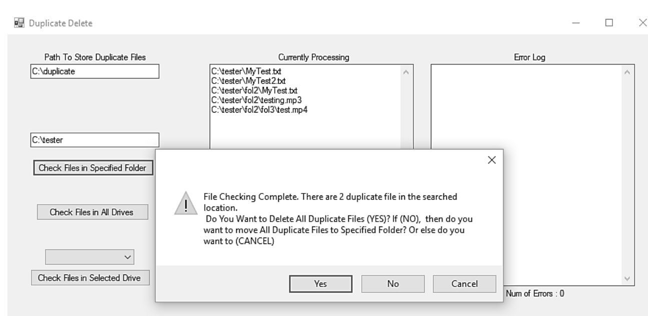

Fig. 9. Screenshot of demo application after analyzing the test file structure

After further testing, it is confirmed that, this also works perfectly with other file types, as well as for selective drive scan and system wide scan.

4.3Time Complexity & Performance

The proposed solution can employ one or more hash functions like SHA-256, MD5, etc. to create an N-layer signature. It is recommended that hash functions that use the same block size be used, so that a block of file can be read just once, and then sent to the hash functions for calculating the hash. This way we can reduce the number of disk accesses required when more than one hash function is used. The time taken for this read operation would grow linearly with the number of files to be scanned and hence, would take O(n) time.

The insertion and search time complexity of a hash table is O(1) as explained before and can be ignored, as it is constant. So, in effect, this solution has O(n) time complexity. Another way to increase the performance of this algorithm is to use multi-threading. Multiple threads can be used to compute the signatures of the different files in parallel.

4.4Applications and future prospect

- In general, this solution can be used to clean-up any system of duplicate files.
- This algorithm can be made to work in real time so that it analyzes the new files added to the system, by means of copy-paste, downloads or just by saving, to check whether any of those files already exist.

  The hash table generated can be saved on the secondary memory so that the N-layer hashes need not be computed over again.
- If this is paired with a full or differential backup solution, then copying redundant files can be avoided, saving a lot of time and space. A custom backup script can be created where it first compares with the existing backed up files, and then only copies those files which are new or have changed since the last backup.
- This process is not memory or processor hungry, so it can also be used in mobile phones or in environments which have limited memory and processing power.
- This solution can be used in online file storages, where, if, multiple users have stored the same file in their personal accounts, then the server will only keep a single copy.
- This algorithm can be further extended to detect similar looking photos and retain only the best one through image processing.

4.5 Drawbacks

Redundant files are sometimes crucial for backup purposes so that they can be successfully restored after a system crash and they should not be deleted. This algorithm is not recommended for such cases.

© 2014- 17, IRJCS- All Rights Reserved

V. CONCLUSION

This paper addresses a major concern with data storage by identifying and managing duplicate files in an efficient manner. This can be further extended with various other solutions as discussed.

REFERENCES

1. "Wikipedia – Hard Disk Drive": https://en.wikipedia.org/wiki/Hard_disk_drive
2. "Difference between: Full, Differential, and Incremental Backup" : http://tinyurl.com/zpnnvv4
3. https://en.wikipedia.org/wiki/Avalanche_effect
4. Pramod George Jose, Soumick Chatterjee, Mayank Patodia, Sneha Kabra, and Asoke Nath, "Hash and Salt based Steganographic Approach with Modified LSB Encoding," International Journal of Innovative Research in Computer and Communication Engineering, vol. 4, issue 6, pp. 10599 – 10610, June 2016.
5. "Wikipedia – Hash Table": https://en.wikipedia.org/wiki/Hash_table
6. "Wikipedia – Associative Array": https://en.wikipedia.org/wiki/Associative_array
7. "Wikipedia – Collision Resistance": https://en.wikipedia.org/wiki/Collision_resistance
8. "Difference between Hashtable and Dictionary": http://tinyurl.com/hljs2lf
9. ".NET data structures": http://tinyurl.com/hqfauaf

Prof. Siladitya Mukherjee has more than 13 years of teaching experience since January, 2004. He worked as a paper-setter and a moderator in Calcutta University and Jadavpur University. He is currently working as an Assistant Professor in the Department of Computer Science, St. Xavier's College (Autonomous), Kolkata. His area of interests includes Web technologies, Unix Operating System, Programming in Object Oriented Technologies and Genetic Algorithm. His research interestsis Data Mining, Artificial Intelligence, Image Processing, Web Technologies and Genetic Algorithms.

Pramod George Jose is a M.Sc. Computer Science graduate from the Department of Computer Science, St. Xavier's College (Autonomous), Kolkata; interested in Cryptography, Steganography, Computer Security and low level working of computer systems.

Soumick Chatterjee is a M.Sc. Computer Science graduate from the Department of Computer Science, St. Xavier's College (Autonomous), Kolkata; interested in Web Technologies, Cross-platform development, Cryptography, Steganography, Machine Intelligence with successful experience in Tech Entrepreneurship.

Priyanka Basak is a M.Sc. Computer Science graduate from the Department of Computer Science, St. Xavier's College (Autonomous), Kolkata; nterested in Cryptography, Image Processing, Machine Learning and Computer Vision.

© 2014- 17, IRJCS- All Rights Reserved