

## Article

# Fingerprint-Based Data Deduplication Using a Mathematical Bounded Linear Hash Function

Ahmed Sardar M. Saeed <sup>1,\*</sup>  and Loay E. George <sup>2</sup>

<sup>1</sup> Information Technology, Technical College of Informatics, Sulaimani Polytechnic University, Sulaymanyah 46001, Iraq

<sup>2</sup> Assistant of University President for Scientific Affairs, University of Information Technology and Communication (UoITC), Baghdad 10011, Iraq; loayedwar57@uoitc.edu.iq

\* Correspondence: ahmed.sardar@spu.edu.iq; Tel.: +964-770-110-5766

**Abstract:** Due to the quick increase in digital data, especially in mobile usage and social media, data deduplication has become a vital and cost-effective approach for removing redundant data segments, reducing the pressure imposed by enormous volumes of data that must be kept. As part of the data deduplication process, fingerprints are employed to represent and identify identical data blocks. However, when the amount of data increases, the number of fingerprints grows as well, and due to the restricted memory size, the speed of data deduplication suffers dramatically. Various deduplication solutions show a bottleneck in the form of matching lookups and chunk fingerprint calculations, for which we pay in the form of storage and processors needed for storing hashes. Utilizing a fast hash algorithm to improve the fingerprint lookup performance is an appealing challenge. Thus, this study is focused on enhancing the deduplication system by suggesting a novel and effective mathematical bounded linear hashing algorithm that decreases the hashing time by more than two times compared to MD5 and SHA-1 and reduces the size of the hash index table by 50%. Due to the enormous number of chunk hash values, looking up and comparing hash values takes longer for large datasets; this work offers a hierarchal fingerprint lookup strategy to minimize the hash judgement comparison time by up to 78%. Our suggested system reduces the high latency imposed by deduplication procedures, primarily the hashing and matching phases. The symmetry of our work is based on the balance between the proposed hashing algorithm performance and its reflection on the system efficiency, as well as evaluating the approximate symmetries of the hashing and lookup phases compared to the other deduplication systems.



**Citation:** Saeed, A.S.M.; George, L.E. Fingerprint-Based Data Deduplication Using a Mathematical Bounded Linear Hash Function. *Symmetry* **2021**, *13*, 1978. <https://doi.org/10.3390/sym13111978>

Academic Editor: Carlo Cattani

Received: 23 September 2021

Accepted: 16 October 2021

Published: 20 October 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

**Keywords:** data deduplication; mathematical bounded linear hashing algorithm; hash lookup; hashing index table

## 1. Introduction

Globally, the amount of digital data is rapidly increasing. Organizations and companies are collecting and storing ever-increasing amounts of data, so more computing power, storage, and network bandwidth are required. According to IDC's Digital Universe Study, the expected quantity of data created in 2010 and 2011 increased significantly from 1.2 zettabytes to 1.8 zettabytes, respectively, and the amount of data produced in 2020 was predicted to be 44 zettabytes [1].

Due to this "data deluge", cost-effectively and efficiently storing and managing the data has become one of the most difficult and important issues in big data and has a significant effect on the overhead cost of storage, processing, and networks. Systems, apps, users, and communication models cause significant redundancy in large chunks of data [2,3]. According to deduplication works carried out via IBM, Microsoft, Intel, Google, and Motorola, approximately three-quarters of digital information is considered redundant; such redundant data consume not only considerable IT resources but also expensive network bandwidths [4]. There is a considerable gap between available storage

capacity and data production for the near future, indicating that much of the information will be lost. As a result, the technology of data deduplication is becoming increasingly necessary to address this problem, and it has emerged as the primary strategy for data reduction [5].

Identifying files with the same size or file name is the most intuitive deduplication approach. However, this method might lead to a deduplicated result that is incorrect. As a result, a hashing-based deduplication approach has been developed to improve accuracy. A hashing calculation and lookup, on the other hand, may lead to a high computing cost [6–8]. File-based deduplication and chunk-based deduplication are two types of data deduplication technology. Although file-based deduplication has a faster throughput, it has a low ratio of duplicated data detection. In comparison to file-based techniques, chunk-based data deduplication finds duplicated data more quickly. As a result, chunk-based deduplication is used by the majority of data deduplication solutions [1].

The three primary stages of the data deduplication process are chunking, hash value production, and redundancy discovery and deletion (lookup and matching). Generally, the method separates ingested files into variable-length or fixed-size chunks; after that, it evaluates a fingerprint regarding each one of the chunks with the use of a cryptographic hash technique (SHA-1 or MD5) and finds duplicate chunks through comparing their fingerprints against a fingerprint store. The chunk is considered a duplicate and is obviated for storage in cases where the fingerprint matches the fingerprint store, indicating that the relevant chunk content has already been saved. If no fingerprint matches, the chunk is considered a distinctive chunk and is written to a distinctive data container [9,10].

As the amount of deduplicated data grows, the performance of data deduplication diminishes. This is due to the fact that the fingerprints' volume is growing exponentially as data grow, and utilizing small chunk sizes causes an increase in the possibility of matching. However, a variety of additional hashes are created, introducing transactional complexity and the possibility of poor performance. Moreover, the chunk's fingerprint values are stored in each one of the entries of the hash index table, which is modified or accessed more often compared to the data chunk container [6,11]. This paper focuses on speeding up the hash generation and redundancy lookup processes. It is focused on the hashing computation overhead caused by the process of allocating each chunk a hash value known as a fingerprint, the memory space needed for storing the hash index table, and the lookup time needed for comparing the fingerprints by determining if a new chunk is considered identical to the one that is already stored.

The following are the contributions of the presented study:

1. A new hash method for the data deduplication process that generates a fingerprint for each chunk is created. The suggested approach applies a mathematical bounded linear hash function to build multiple hashes that require fewer computational resources and minimize the hash index table's size based on its symmetrical characteristics, whereas content-defined chunking with the use of MD5 or SHA-1 for data fingerprinting consumes significant processing resources for calculating hash values.
2. For reducing hash judgements, a novel heretical hash lookup framework is created. Through removing the byte-to-byte comparison that is required for comparing chunks, the cascade comparison is going to decrease the time required to compare them.

The remainder of this work goes into the specifics of the suggested system. The works in Section 2 are linked to data deduplication. The technique of the system is detailed in Section 3. The suggested system is outlined in Section 4. The results of the proposed approach are indicated in Section 5. The final portion concludes the paper and discusses future projects.

## 2. Related Works

For the past few years, the systems of data deduplication have been the topic of extensive research. Fingerprint lookup and generation has become a bottleneck, restricting the throughput and scalability of data deduplication systems [1,6,12], as the volume of

data has increased dramatically. Several previous studies have contributed to this topic in various ways. Different approaches are offered for reducing the overhead related to fingerprint lookup and generation. Benjamin Zhu et al. [6] presented three techniques of data deduplication to alleviate the disk bottleneck: the first is the Summary Vector, which is considered a compact data structure in memory used for identifying new chunks; the second is the Stream-Informed Segment Layout, which uses a new data layout model for sequentially accessing chunks in disk; and the third is Locality Preserved Caching, which maintains high cache hit ratios through improving the locality related to duplicate chunks' fingerprints. All three strategies together remove 99% of the disk access for the deduplication system, resulting in a single-stream throughput of 100 MB/s and a multi-stream throughput of 210 MB/s. Deepavali et al. [13] proposed Extreme Binning as a scalable and parallel deduplication method that takes advantage of file similarity rather than locality and uses only one disk access for each file for chunk lookup, resulting in a satisfactory throughput. To improve performance, the system has scaled out multi-nodes dependent on the amount of input data. For parallelized data management with little overhead, each file is assigned to only one node with the use of a stateless routing technique. The system divides the hash index table into tiers, with no data or index sharing between nodes, leading to a small memory footprint and allowing the system to sustain throughput for a large dataset compared to a flat hash index scheme. In addition, the loss of deduplication is minor, and the advantages in memory utilization and system scalability readily compensate for it. To alleviate the chunk-lookup disk bottleneck that fingerprint-based deduplication systems confront, Lillibriged et al. [14] employed sampling and sparse indexing that uses inherent locality. For determining which chunks have previously been saved, they preserve a small fraction of the sampling index in memory, which needs only a few seeks per chunk, rather than a full index that indexes every chunk. The system utilizes sampling and a sparse index to divide an incoming stream into reasonably large chunks and deduplicate each piece against only some of the most similar prior chunks. The suggested technique reduces the average number of champions loaded per chunk by 3.9% while increasing the deduplication factor by 1.1%.

Guanlin et al. [15] developed BloomStore, a flash-based store architecture that enables a probabilistic duplication lookup and ensures a very low amortized memory overhead by keeping a flash-page-sized data buffer and an extremely small buffer per BloomStore instance in memory. It also stores the entire index structure on a flash disk for improving the deduplication performance by achieving a high lookup/insertion throughput through decreasing the maximum number of flash page reads with a key range partitioning. The BloomStore design reached a considerably better key lookup throughput with a 22.5% lower memory usage through decoupling the chunk existence and location searches to avoid unwanted complex searches when a hash is not stored. Liangshan Song et al. [7] worked on the hash lookup stage by providing a fingerprint-prefetching approach that makes use of data locality and file similarity via detecting comparable files that share a significant number of identical data blocks with the use of a comparable file recognition algorithm. Moreover, to optimize the cache hit ratio, the fingerprints are organized on the disk depending on the sequence of data streams. As memory is insufficient to hold the entire hash index table, a portion of it will be stored on the disk and requested or recalled to be in memory before they are actually required, demonstrating an adequate optimization for fingerprint lookup and leading to the conclusion that a fingerprint prefetching algorithm is more efficient for large files than for small files.

Naresh et al. [16] suggested a bucket-based data deduplication method in which the hash index table is divided into buckets, and MapReduce is used to locate duplicate chunks with the use of the Hadoop Distributed File System (HDFS). First, the system will create fixed-size chunks before making fingerprints for such chunks by means of the MD5 hashing algorithm and storing hash values in corresponding buckets. For detecting duplicate MapReduce models, the system will compare such hash values with previously stored hash values in the bucket storage. When such hash values already exist in bucket

storage, they will be indicated as a duplicate. The solution attempts to address the issue of a large hashing index table and hash lookup through storing hashes in distinct buckets to be accessible via MapReduce. In comparison to the present fixed-size chunking techniques, the suggested approach hash time and chunk lookup is minimal. Shrivastava et al. [17] developed a modified hash value concept that uses the SHA-2 hashing algorithm rather than the usual MD2, SHA-1, and MD5 hashing algorithms to improve the deduplication system efficacy, particularly at the hash creation stage. With regard to the time and quantity of secure hashes created, the suggested technique provides enhanced security and takes less time compared to the MD5, MD2, and SHA-1 approaches, although the size of the fingerprint formed by SHA-2 is larger than that of the existing hashing algorithms. As a result, the hashing stage's throughput was increased, yet the problem of the hashing index table's size was not resolved. Dongyang et al. [18] proposed hardware-based deduplication acceleration using a FPGA-based accelerator interface and three compute-intensive kernel modules. The expected results were achieved by 30% enhancement in the deduplication ratio and six times latency improvement compared to software-based deduplication.

The goal of the research mentioned above is to figure out how to decrease the number of chunk hashes in memory while also speeding up the table lookup procedure. There are two primary forms of optimizations for lowering the size of the hash index table in memory: decreasing the hash length and employing a large chunk size, although both have drawbacks. More hashes will be recorded for various chunk blocks as the number of chunks increases, resulting in a larger hash index table. In addition, the chunking algorithm might be utilized to reduce the overall number of hashes in memory by using a large average chunk size. This method, on the other hand, can lower the deduplication ratio. Table 1 shows different data deduplication research studies.

**Table 1.** Related works compression table.

Research Paper	Technique Used	Achievement	Limitation
Avoiding the disk bottleneck in the data domain deduplication file system [6]	Summary Vector Stream-Informed Segment Layout Locality Preserved Caching	Enhancing the chunks fingerprint lookup by reducing the disk access	Disk bottleneck had been reduced but still the hashing index table size is big and can't be hosted in memory
Extreme binning: Scalable, parallel deduplication for chunk-based file backup [13]	Extreme Binning Scaled out multi-nodes system	Reduce the fingerprint index table size by divides into tiers for better memory utilization	Multi-nodes system add complexity which affect deduplication throughput
Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality [14]	Sampling and sparse indexing for the hashing index table	Preserve a small fraction of the hashing index table in memory reduce the hashes lookup time	Deduplication factor enhanced by 1.1% only as the big part of the hashing index table is still in disk
BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash [15]	BloomStore, a flash-based store architecture	Store the entire hash index table on a flash disk achieving a high lookup throughput and lower memory usage	Bloom-filter based on flash disk will add a high cost impact to the system as its considered as hardware-based deduplication
Exploiting fingerprint prefetching to improve the performance of data deduplication [7]	Fingerprint-prefetching approach using locality and file similarity	Fingerprints are organized on the disk depending on the sequence of data streams which optimize the prefetching cache hit ratio	Fingerprint prefetching algorithm is more efficient for large files than for small files.
Bucket based data deduplication technique for big data storage system [16]	Bucket-based and MapReduce under Hadoop Distributed File System (HDFS)	Storing hashes in distinct buckets to be accessible via MapReduce reduce hashing time and chunk lookup	Fixed-size chunking reduce deduplication ratio

Table 1. Cont.

Research Paper	Technique Used	Achievement	Limitation
A Big Data Deduplication Using HECC Based Encryption with Modified Hash Value in Cloud [17]	SHA-2 hashing algorithm	Enhance hashing time and throughput	Problem of the hashing index table's size has not been resolved due to size of the fingerprint formed by SHA-2
Hardware accelerator for similarity based data dedupe [18]	Hardware-based deduplication acceleration using FPGA-based accelerator interface	Enhancement in deduplication ratio and reduce latency compared to software-based deduplication	Higher cost due to the required investment in acceleration hardware

### 3. Methodology

Data deduplication is the process of eliminating redundant data through storing just the data that is unique. When many copies of the same dataset must be stored, this strategy efficiently minimizes the storage capacity requirements. Chunking, fingerprinting, and lookup and matching are the three stages of the deduplication process [19,20]. In this paper, we will focus on the last two stages by proposing a new fingerprinting (hashing) algorithm and efficient lookup procedure.

#### 3.1. Fingerprinting (Hashing) Stage

The most difficult aspect of deduplication systems is swiftly identifying duplicate data segments. Byte-by-byte comparison is impossible, since it necessitates too much I/O. As a result, the majority of deduplication systems use “fingerprints” for detecting duplicate data segments. The fingerprints must satisfy the requirement that two fingerprints are identical if and only if the two data segments corresponding to them are identical. A conventional cryptographic hash algorithm is used to establish a unique identification referred to as a fingerprint for each chunk [11,21]. A hash algorithm is considered as a function that converts an arbitrary-length input into a fixed amount of output bits, known as the hash value. Each time the same input is hashed, the hash value must be the same. The resulting fingerprint (hash) values are saved in a hash index table. In addition, the fingerprint of incoming data is compared to the ones already recorded in the hash index table throughout the deduplication process. When the fingerprint has already been stored, the incoming data are replaced with a link to it. Furthermore, the data are written to the disk as a new unique chunk if the fingerprint does not exist [22,23]. In a hash-based deduplication system, duplicate data are identified by comparing the identifier of the data, so the hash function should be strongly efficient and collision-resistant. The drawback is that, for a large dataset, the size of a hash index table grows dramatically in order to accommodate all the fingerprints by allocating a unique index position for every fingerprint that needs to be stored. In such a case, the hash index table size grows out of the memory range and requires paging and other operating system-based operations. This likely results in an overall reduced performance, as fingerprinting and lookup are the determining factors of data deduplication with which the data are identified as unique or duplicate [4]. Cryptographic hash functions such as SHA-1 and MD5 are the most commonly used methods for this purpose [1].

##### 3.1.1. The MD5 Hashing Algorithm

Ron Rivest of MIT invented the Message Digest 5 (MD5) cryptographic hash function, which contains a string of digits. It is essentially a more secure version of his earlier technique, MD4, which is faster than MD5. Its purpose is to protect the data's integrity and identify all the changes made to the data. Through compressing a message of any length, MD5 creates a fixed 128-bit hash value [24]. Message blocks go through four rounds of processing. The MD5 hash algorithm aids in data integrity verification. MD5 is a frequently utilized strategy for identifying, protecting, and certifying data in digital signature and cryptography applications. MD5 is a text compression algorithm that divides

text into 512-bit blocks, which are afterward fragmented into 16 32-bit sub-blocks. The MD5 encryption algorithm produces a series of four blocks, each of which is 32 bits in size, resulting in a hash value of 128 bits [25].

### 3.1.2. The SHA-1 Hashing Algorithm

The National Institute of Standards and Technology (NIST) developed SHA-1 as a security algorithm according to the SHA algorithm's patch results. The major hash algorithm is SHA-1, which is based on the MD4 method [26]. To construct a 160-bit message digest, SHA-1 takes the bits of the message's maximum length. This arithmetic operation, like the MD5, is divided into 32-bit words with a 512-bit length for processing units, with four loop operators and 20 rounds in each loop, for a total of 80 rounds. SHA-1, on the other hand, is a complicated method using data shifting and logical functions [25,27].

The SHA-1 encryption algorithm has the advantage of being stronger, but the time required for encryption is slow in comparison to the MD5 algorithm, as MD5 has 64 iterations, whereas SHA-1 has 80 iterations, so SHA-1 seems to be slower compared with MD5. Practically, SHA-1 will require a larger storage size compared to MD5, so the MD5 algorithm is cheaper to compute and utilizes less disk space [28]. Both MD5 and SHA-1 are computationally demanding; in specific applications, chains of millions of hash algorithm rounds are evaluated. One of these applications is data deduplication, in which the chunk hash calculation forms a bottleneck in the form of processor utilization. Using a faster hash function is therefore attractive.

### 3.1.3. The Mathematical Bounded Linear Hashing Algorithm

The mathematical bounded linear hashing algorithm creates a hash value by multiplying a random sequence of numbers by the linear bounded sum of a string of nonrepeatable zero bytes. The employment of various random number sequences to generate various short hash values is sufficient to generate different signatures for characterizing the plain text contents of chunks. In addition, the mathematical signatures are hash functions with algebraic characteristics identifying large data objects with a low probability of collision. Moreover, the mathematical operations utilized for computing the hash values are simple, primitive mathematical operations, resulting in a computing overhead that is extremely low compared to conventional security hash functions. Compared to the computational complexity related to cryptographic hash functions (such as MD5 and SHA-1), it has a low computational cost. The use of more than one hash sequence to describe the string content might also result in a combined value with lower collision rates while improving the lookup stage. The main objective to using a mathematical hash function is that there are many properties and features that strongly required, cryptographically secure hash functions, which are not required for non-cryptographically secure hash functions [29,30].

### 3.1.4. Hash Collision

A hash collision is a situation where chunk fingerprints coincide and the chunks themselves differ. The result of a collision in a deduplication system is that the incoming file is not stored correctly, rendering it essentially unusable. A chunk collision therefore always constitutes data loss [31].

When hash collision is not handled efficiently, the size of a hash index table grows dramatically to accommodate all fingerprints by allocating space for every fingerprint that needs to be stored. In such a case, the table size grows out of memory range and requires paging and other operating system-based operations. This likely results in an overall reduced performance, as fingerprint indexing and lookup are the determining factors of data deduplication, with which the data are identified as unique or duplicate. If there is a hash collision—two different chunks have the same hash value—then the system has lost the newer data [15,32]. We can easily control the hash collision probability by using good hash functions with more bits, replacing MD5 with 128 bits with SHA-1 (160 bits) or upgrading to a member of the SHA-2 family, such as SHA-256 with 256 bits, but in such a

case, the hashing index table size grows and leads to system performance degradation in both hash generation and matching stages [9].

### 3.2. Lookup and Matching Stage

The fingerprint of split blocks is compared to the fingerprint table at this stage. When a block in the table has a duplicate, it is not written to disk; only the metadata are stored. If there is no duplicate regarding the block in the table, the block's fingerprint is entered into the hash index table as a unique block in the system. This stage includes an additional fingerprint indexing procedure to check for duplicate material and updates the file's metadata, which includes the pointer-based mapping of blocks and files. Moreover, hash indexing is a crucial step in the fingerprinting process, as it organizes fingerprints into a logical order. It is used to verify the uniqueness of data chunks before storing or discarding them [33]. As the index size is proportional to the data size, the fingerprint (hash) index is a design problem for scale and resource consumption in a conventional deduplication system. Conserving memory and disk IOPS resources is critical in a primary data deduplication system. To link fingerprints to physical addresses of chunks or files, a fingerprint index must be created. Checking for the fingerprint of a duplicate file or chunk in the fingerprint index can help identify it. There are two types of approaches for detecting duplicate fingerprints. (1) Exact detection compares each newly arrived fingerprint against the whole fingerprint index. This technique seeks to discover all duplicate data by checking all fingerprints in the index store extensively. This approach has a significant flaw in terms of fingerprint query performance, as the large search space might result in unnecessary disk visits. (2) In near-exact detection, each fingerprint only checks a portion of the fingerprint index at the risk of losing certain deduplication opportunities, reaching an approximate deduplication ratio, such as the ratio between the original total data size and the amount of data that is actually stored following deduplication; this approach represents data deduplication efficiency and is a significant indicator when comparing various deduplication systems [7]. When performing data deduplication, fingerprints are employed for describing and identifying identical data blocks. However, as the volume of data expands, so does the number of fingerprints. The fingerprints must be saved on hard drives because of the limited memory size. When the fingerprints are not satisfied in memory, disk I/Os are created to obtain the fingerprints from the disk. Due to the random and small I/Os, data deduplication performance suffers considerably. Moreover, the chunk-lookup disk bottleneck problem, which limits the throughput and scalability of deduplication-based storage systems, is a popular challenge. It is impractical to keep a large index in RAM with a considerable amount of data, and a disk-based index with one seek for each incoming chunk is far too slow for large-scale storage [6,15].

### 3.3. Experimental Datasets

For testing the system efficiency and performance, three datasets with different features were used. The first dataset contains various versions of Linux source codes from the Linux Kernel Archives [34], second dataset contains 309 versions of SQLite [35], and the third dataset contains Oracle RMAN Backup. The features of the datasets used are listed in Table 2.

**Table 2.** The characteristics of the tested datasets.

Dataset	Dataset 1	Dataset 2	Dataset 3
Dataset Name	Linux Kernel	SQLite	Oracle RMAN Backup
Dataset Type	Linux source codes (3.16.57-5.5-rc4)	309 releases of SQLite from version 1.0 to 3.33.0	Backup
No. of Files	926,953	212,741	36
Dataset Size (in MB)	11,161.6	6594.6	19148.8
Dataset Size (in GB)	10.9	6.44	18.7

#### 4. The Proposed System

The most prevalent methods of data deduplication include comparing chunks of data to find duplicates. This method is known as hash-based or fingerprints-based deduplication, and it involves the following steps: (1) chunking, where deduplication begins with the data being split into data blocks, referred to as chunks; (2) computing fingerprints, where, for each chunk, its fingerprint is computed using the hash function; and (3) duplicate detection by matching lookups, where the new chunks are compared with the chunks that have already been stored in the system. If the fingerprint (hash value) of the chunk is found in the hashing index table, the chunk will be deleted, and a logical reference is added to the matched one. If not, then it is considered a new chunk; the location and the hash value of the chunk are stored in the database and in the hashing index table.

Though deduplication can be a promising solution, it suffers from the wasted time required for generating the fingerprints and lookup for duplicate chunks, and this is too expensive to be effective in terms of its computation and space overheads. In a hash-based deduplication system, a considerable amount of time is spent on both fingerprint generation and lookup, as it frequently needs extra CPU processing for intensive hash calculations, as well as extra disk I/O processing to compare the fingerprints. It can be saved by efficiently handling these two stages.

This work presents a data deduplication design with the following characteristics: We effectively introduce a mathematical multi-hash function that enhances the hashing stage throughput and reduces the size of the fingerprint index. Performance degradation is minimized by enhancing the hash lookup procedure to effectively support data deduplication. The next guiding requirements guided the creation of our hashing function are as follows: (1) The hashing and matching time must be substantially short compared to old approaches due to a high throughput and low latency. (2) As short strings are the primary unit for computing fingerprints, fingerprints must not exceed 80 bytes in length, resulting in storage savings in the hash index table. Given that shorter fingerprints may have a higher collision rate, the purpose of this section is to suggest a hash function that performs better and has the fewest hash collisions.

In the matching stage, where there are many queries for checking the deduplication fingerprint, the lookup process could be time-intensive; thus, this work utilized a hierarchical multilevel matching mechanism to optimize the lookup process, since the matching stage requires lookup for the entire hash index table to find match fingerprints. This phase could be a time-consuming process that needs a heavy I/O request. The idea is to find a simple method to test whether a data fingerprint exists in the index table.

##### 4.1. Multi-Hash Function Using a Mathematical Bounded Linear Hashing Algorithm

The collision problem affects the legacy deduplication systems, which require a substantial amount of disk space, processing power, and time to solve. A novel hashing approach is suggested in this paper for saving resources and reducing processing time. To compute multi-hash values for each one chunk, the system's hashing stage employs a new multi-hash function using a mathematical bounded linear hashing algorithm. As a result, each chunk generated by the chunking step will be given to the suggested hash function, which will create five hash values for describing the chunk contents. Moreover, the hash value is generated by multiplying a random sequence of numbers by a mathematical linear bounded sum of nonrepeatable zero bytes. The employment of various random number sequences to generate various short hash values is sufficient to generate various signatures for characterizing the plain text contents of chunks. As the hash values are computed using a primitive, simple mathematical operation, the needed overhead of computing the hash is extremely low compared to the computational complexity of typical cryptographic hash functions (such as MD5 and SHA-1). As a result, utilizing more than one hash sequence for representing the string content might result in a combined value with lower collision rates. In addition, a mathematical function with a size of 16 bits generates each hash. A total of 80 bits are required for storing five hashes, as each hash is 2 bytes (16 bits). Conventional

hashing functions (MD5 and SHA-1), which are utilized by other content-defined chunking approaches, demand a substantial amount of computational resources for calculating hash values and of storage space for storing them. The number of bits required for saving the suggested hashes is smaller than the number of bits required for saving SHA-1 (160 bit) and MD5 hash values (128 bit). The hashing stage is divided into two phases:

- The first phase is the initialization: In this phase, five hash buffers will be initialized with different values, and each buffer consists of 255 elements. Different random values will be generated between one and 255, and these are used as an index to permuted the values in the hash buffer. The resulting five buffers will be used in the next stage to generate the five hashes for each chunk. The steps of this phase are illustrated in Algorithm 1.
- The second phase is the multi-hash generation function: In this phase, the resulting five hash buffers from the initialization phase with the chunk data will be used as an input to generate five hash values for each chunk using mathematical operations. The steps of this stage are described in Algorithm 2, and Figure 1 illustrates it in a flowchart.

---

#### Algorithm 1 Initialization Phase

---

<b>Objective:</b>	Initialize five hash buffers
<b>Input:</b>	I as Integer
<b>Output:</b>	Five hash buffers as: HF1: array of integer contains 255 random values HF2: array of integer contains 255 random values HF3: array of integer contains 255 random values HF4: array of integer contains 255 random values HF5: array of integer contains 255 random values
<b>Step1:</b>	Fill the first hash buffer (HF1) with 255 numbers For I = 0 to 201 Do HF1[I] = I + 1 End For
<b>Step2:</b>	Generate random values for the first hash buffer (HF1) For I = 201 to 1 Do Generate random value between 1 and 255 Do permutation on array HF1 End For
<b>Step3:</b>	Fill the second hash buffer (HF2) with 255 numbers For I = 0 to 201 Do HF2[I] = I + 6 End For Generate random values for the second hash buffer (HF2) For I = 201 to 1 Do
<b>Step4:</b>	Generate random value between 1 and 255 Do permutation on array HF2 End For
<b>Step5:</b>	Fill the third hash buffer (HF3) with 255 numbers For I = 0 to 201 Do HF3[I] = I + 10 End For

---

**Algorithm 1** *Cont.*


---

	Generate random values for the third hash buffer (HF3) For I = 201 to 1 Do Generate random value between 1 and 255 Do permutation on array HF3 End For
<b>Step6:</b>	
	Fill the fourth hash buffer (HF4) with 255 numbers For I = 0 to 201 Do HF4[I] = I + 10 End For
<b>Step7</b>	
	Generate random values for the fourth hash buffer (HF4) For I = 201 to 1 Do Generate random value between 1 and 255 Do permutation on array HF4 End For
<b>Step8:</b>	
	Fill the fifth hash buffer (HF5) with 255 numbers For I = 0 to 201 Do HF5[I] = I + 10 End For
<b>Step9</b>	
	Generate random values for the fifth hash buffer (HF5) For I = 201 to 1 Do Generate random value between 1 and 255 Do permutation on array HF5 End For
<b>Step10:</b>	

---

**Algorithm 2** Multi-Hash Generation Phase

---

<b>Objective:</b>	Generate five unique hash values for each chunk Chunk as array of bytes Chunk Length
<b>Input:</b>	HF1: array of integer contains 255 random values HF2: array of integer contains 255 random values HF3: array of integer contains 255 random values HF4: array of integer contains 255 random values HF5: array of integer contains 255 random values
<b>Output:</b>	Five hash values for the chunk
<b>Step1:</b>	Initialization Hash1 $\leftarrow$ 3, Hash2 $\leftarrow$ 37, Hash3 $\leftarrow$ 17, Hash4 $\leftarrow$ 31, Hash5 $\leftarrow$ 51, Li $\leftarrow$ 0
<b>Step2:</b>	Compute five hash values for the chunk For I = 0 to Chunk Length-1 Do Li = Li + 1 If Li > 255     Li = Li - 255 Hash1 = Hash1 + (HF1[Li] * Chunk[I]) If Hash1 > 65,535     Hash1 = Hash1 and 65,535 Hash2 = Hash2 + (HF2[Li] * Chunk[I]) If Hash2 > 65,535     Hash2 = Hash2 and 65,535 Hash3 = Hash3 + (HF3[Li] * Chunk[I]) If Hash3 > 65,535     Hash3 = Hash3 and 65,535 Hash4 = Hash4 + (HF4[Li] * Chunk[I]) If Hash4 > 65,535     Hash4 = Hash4 and 65,535 Hash5 = Hash5 + (HF5[Li] * Chunk[I]) If Hash5 > 65,535     Hash5 = Hash5 and 65,535 End For

---

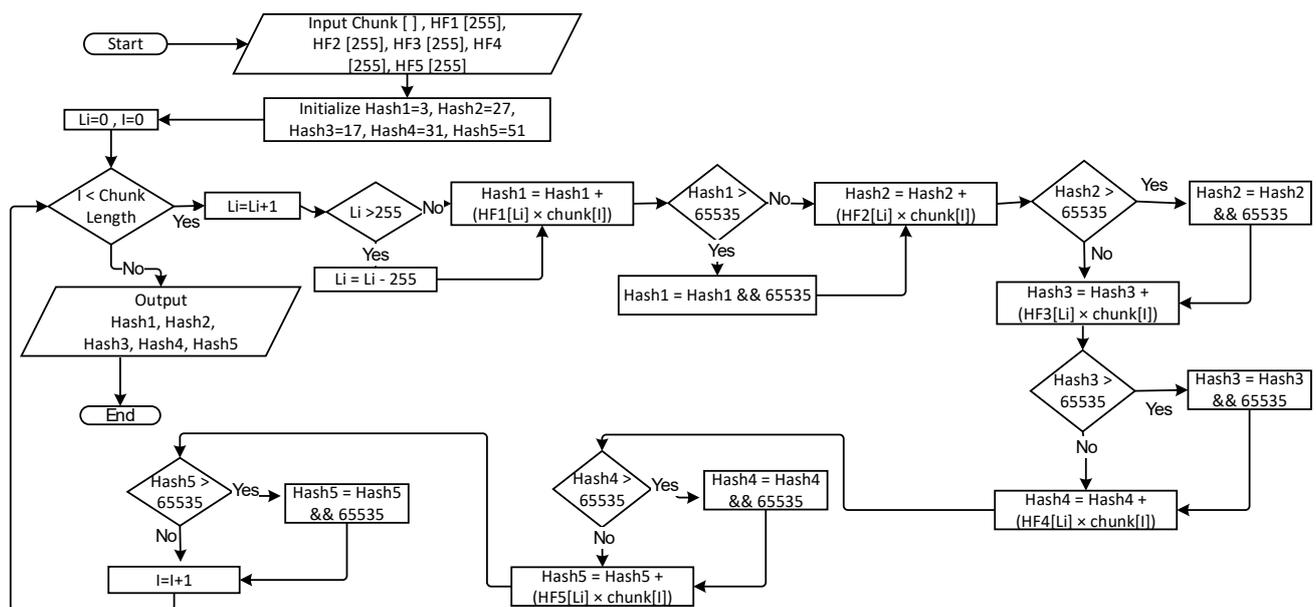


Figure 1. Multi-hash algorithm flowchart.

#### 4.2. Lookup and Matching Stage

The fingerprint lookup module is essential to a deduplication system, but it can become time-intensive as the fingerprint table grows in size. When the file passes the two preview stages (hashing and chunking), rather than comparing the entire chunk byte for byte with all chunks in the database, each data chunk will have its hash value, and the deduplication process will determine whether there is a match or not through comparing the hash value regarding the new chunk with the hash of previously stored data. When a match is detected, the new data will not be saved again because it has already been saved, yet it is going to be replaced by a reference to the name of the previously saved chunk. When no match is detected, such chunks are clearly distinctive, and no byte-to-byte comparison is required. Through comparing only chunk fingerprints instead of the entire chunks, this approach saves time during the matching step. Conventional systems' matching strategies are depicted in Figure 2. Due to the way data are chunked and hashed, and to how the hashes are compared for determining which chunks can be deduplicated, the process of fingerprint lookup has become a bottleneck, which limits the throughput and scalability of deduplication-based storage systems. Identical hashes might occur when two distinct bits of data are combined, and this is referred to as a hash collision. Simply put, the hash collision problem occurs when two separate datasets generate the same hash. When a hash collision takes place during data deduplication, two completely separate datasets will be detected as two identical datasets, resulting in data loss.

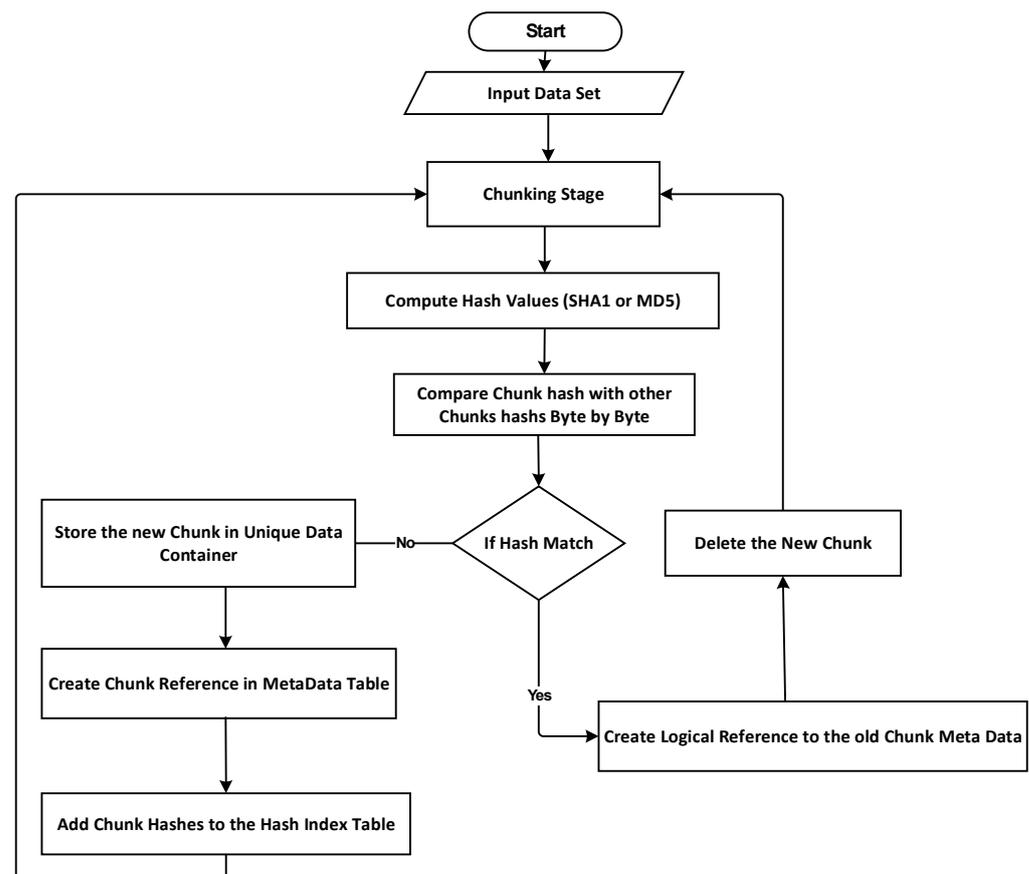


Figure 2. Lookup and matching stage flowchart.

In the proposed hierarchical multi-level lookup and matching technique, a new file will be chunked, and each chunk is compared based on its size. If the chunks have the same size, then the first hash will be compared with the first hash of stored chunks in the hashing index table. If there is a match, then the second, third, fourth, and fifth hashes are compared. If the five hashes match, then a match occurs, and there is no need for a byte-to-byte comparison. This step will save the time needed to compare large-sized hashes such as MD5 or SHA-1 byte to byte; instead, the system will compare only five small numbers one by one to accelerate the process of the fingerprint lookup stage by simplifying the hash judgement, considering that fingerprint lookup latency is greatly impacted by the number of fingerprints and the size of each fingerprint. As comparing five tiny values is faster than comparing the entire hash byte by byte, the test result demonstrates a considerable enhancement in the time required by the matching process.

Figure 3 illustrates the matching strategies of the proposed system with the hierarchical multi-level hashing technique.

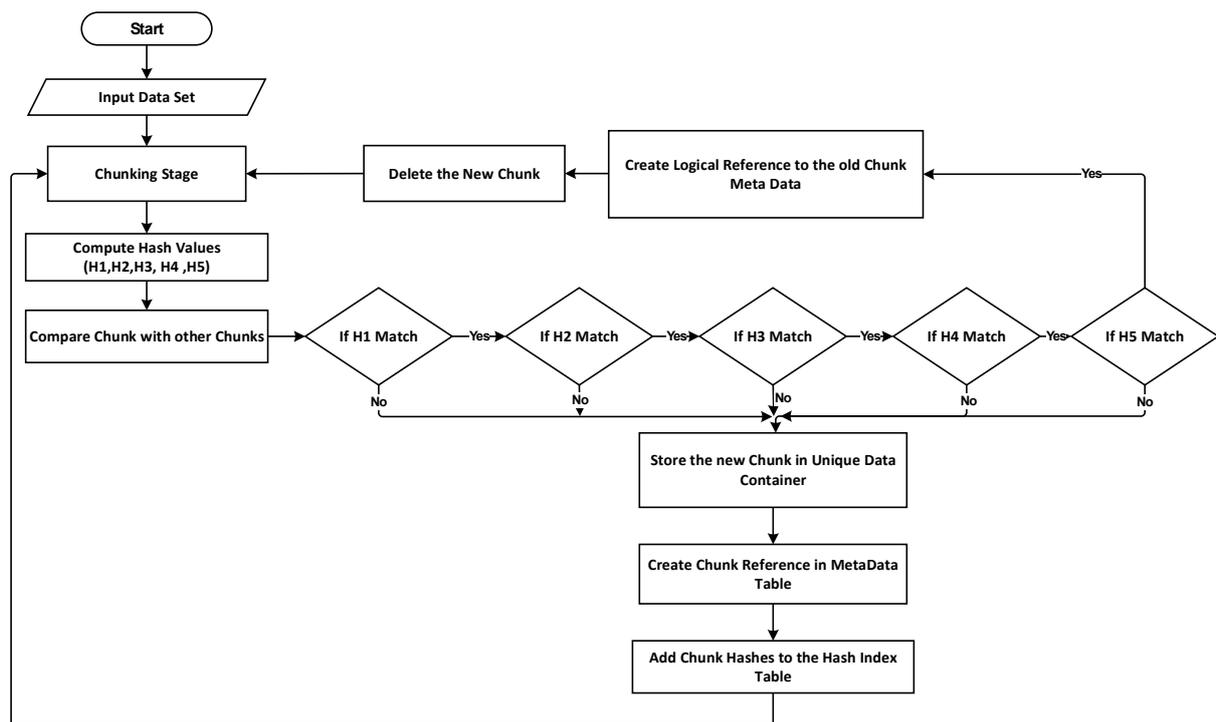


Figure 3. Lookup and matching stage flowchart.

## 5. Experiment

### 5.1. Experimental Setup

We used the C# programming language to create the proposed system. The following is a description of the computer configuration used in the experiment:

- RAM: 16 GB DDR3;
- CPU: Intel Core i7-3820QM @ 2.70 GHz 4-core processor;
- Operating System: Windows 10 64-bit;
- Disk: 1TB PCIe SSD.

### 5.2. Experimental Results

In this work, we carried out comprehensive tests with both real-world and synthetic datasets to evaluate the proposed system's performance. To produce a highly dependable result throughout the testing and implementation phase, the suggested technique was examined with three datasets: Linux Kernel, SQLite, and Oracle RMAN Backup.

The goals of our experimental evaluation were to measure the hashing index table size, hashing time, hashing throughput, matching time, and number of collisions. Since we tested three kinds of datasets, we measured the effects of the proposed system on these datasets. For our evaluations, we primarily used storage saving and throughput as the key performance indicators. Each test had identical experimental circumstances and was repeated five times to eliminate any disparities.

#### 5.2.1. Hashing Index Table Size

The size of the fingerprint index table is an important metric for system overhead in deduplication, which significantly affects the system scalability. A comparison between the suggested mathematical multi-hash function and the SHA-1 and MD5 hash functions showed that it utilizes a smaller storage size.

Figure 3 shows the hashing index table size compared with SHA-1 and MD5. It is significantly reduced due to the smaller hashes used as chunk fingerprints.

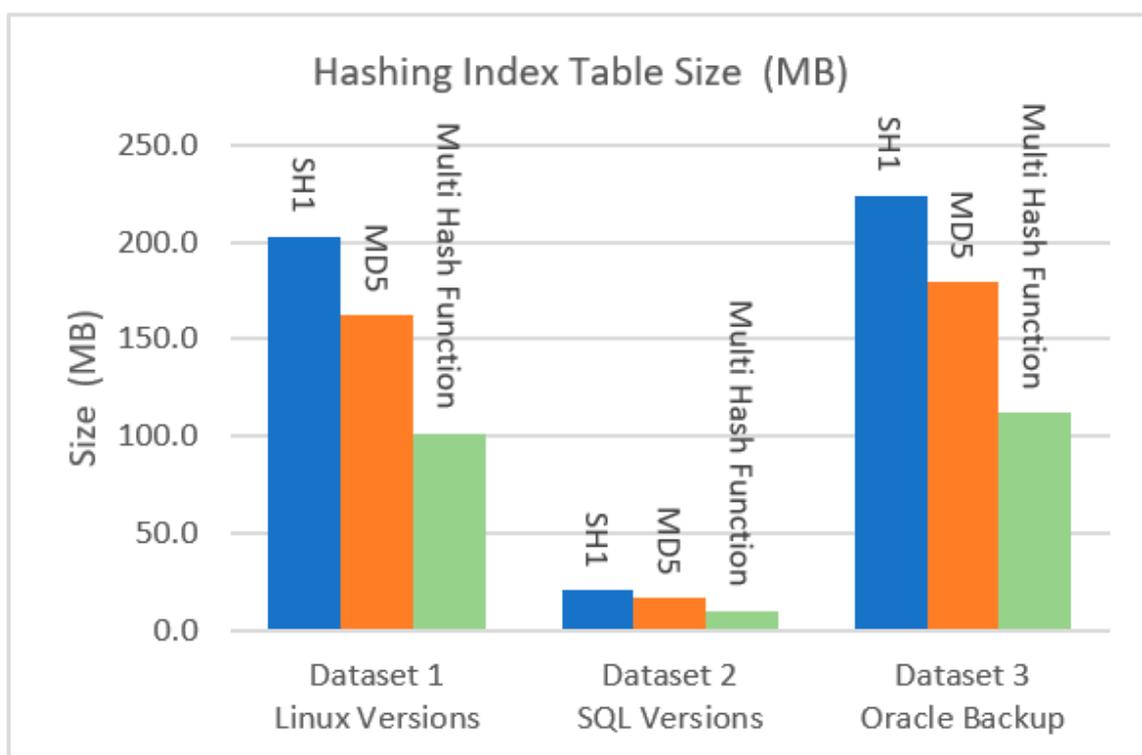
The effect of the hash algorithms on the storage size needed to store fingerprints in the index table was determined by Equation (1) and is shown in Table 3 and Figure 4.

Following deduplication, the total number of chunks for Dataset 1 was 10,626,993, was 1,066,606 for Dataset 2, and was 11,749,768 for Dataset 3.

$$\text{Hashing index table size} = \text{No. of Bytes per Fingerprint} \times \text{Total No. of Chunks after DD} \quad (1)$$

**Table 3.** Required storage size for storing fingerprints (hashing index table size).

	Hashing Index Table Size (MB)		
	SH1	MD5	Multi-Hash Function
Dataset 1 (Linux Versions)	202.7	162.2	101.4
Dataset 2 (SQLite Versions)	20.3	16.3	10.2
Dataset 3 (Oracle Backup)	224.11	179.29	112.05



**Figure 4.** Hashing index table size comparison.

### 5.2.2. Hashing Time and Throughput

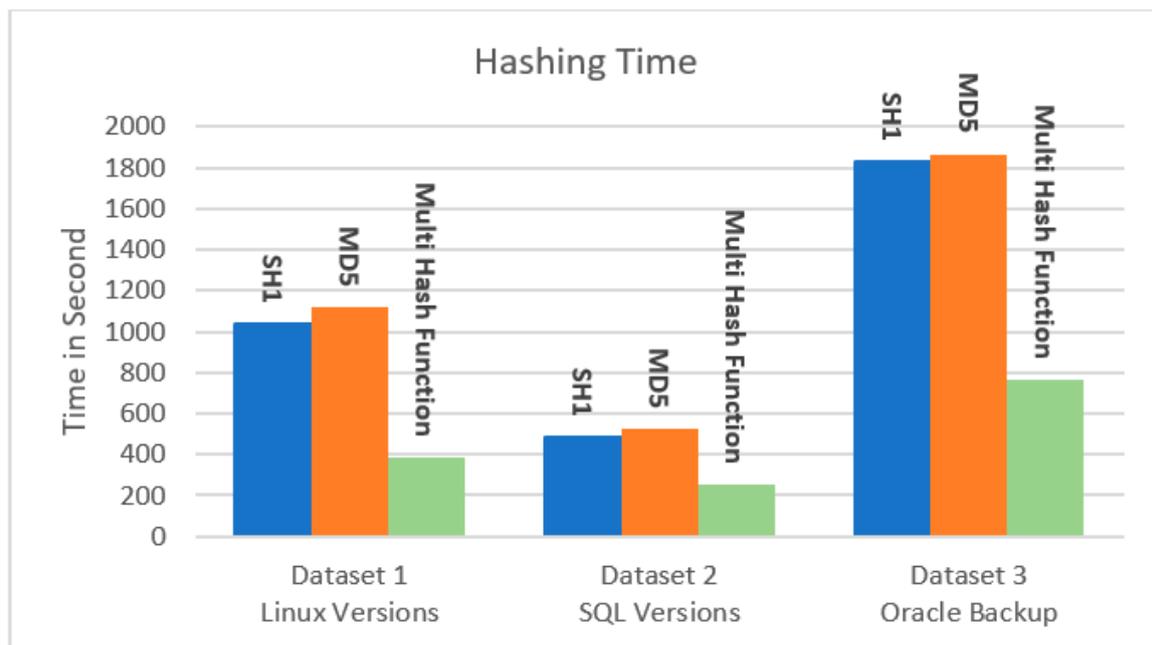
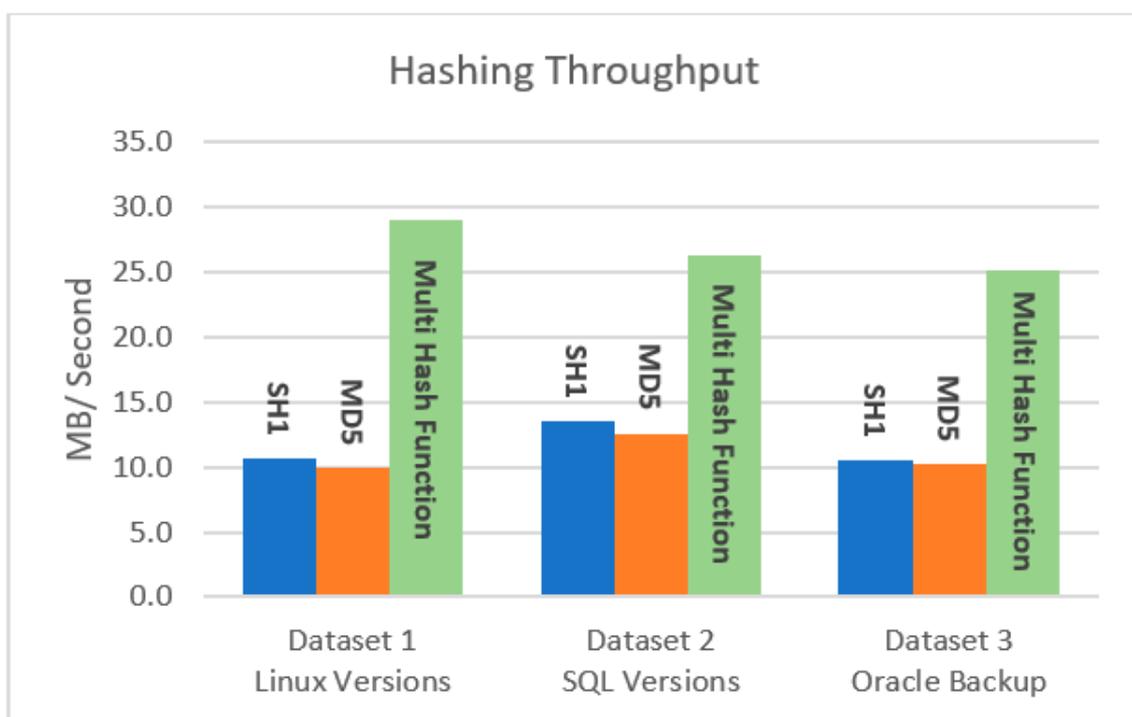
Another main issue is computing the execution time of the processing algorithms and checking the efficiency in terms of their working capability. This approach provides high-speed processing in this area and generates a hash value for the input chunks in less time. Figure 5 shows that our hashing algorithm outperforms all the other hashing techniques.

Table 4 and Figures 5 and 6 show the effect of the hash algorithms on the time and throughput of the hashing stage, as calculated by Equation (2). Based on the results below, the suggested technique takes the least amount of hashing time for all three datasets, resulting in a higher throughput than the conventional hashing algorithms.

$$\text{Hashing Algorithm Throughput} = \text{Hashing Computational overhead} = \frac{\text{Processed Data in MB}}{\text{Time in Second}} \quad (2)$$

**Table 4.** Hashing time and throughput for the proposed multi-hash function compared with SHA-1 and MD5.

	Hashing Time (S)			Hashing Throughput (MB/s)		
	SH1	MD5	Multi-Hash Function	SH1	MD5	Multi-Hash Function
Dataset 1 (Linux Versions)	1040	1121	385	10.7	10.0	29.0
Dataset 2 (SQLite Versions)	485	524	251	13.6	12.6	26.3
Dataset 3 (Oracle Backup)	1831	1866	763	10.5	10.3	25.1

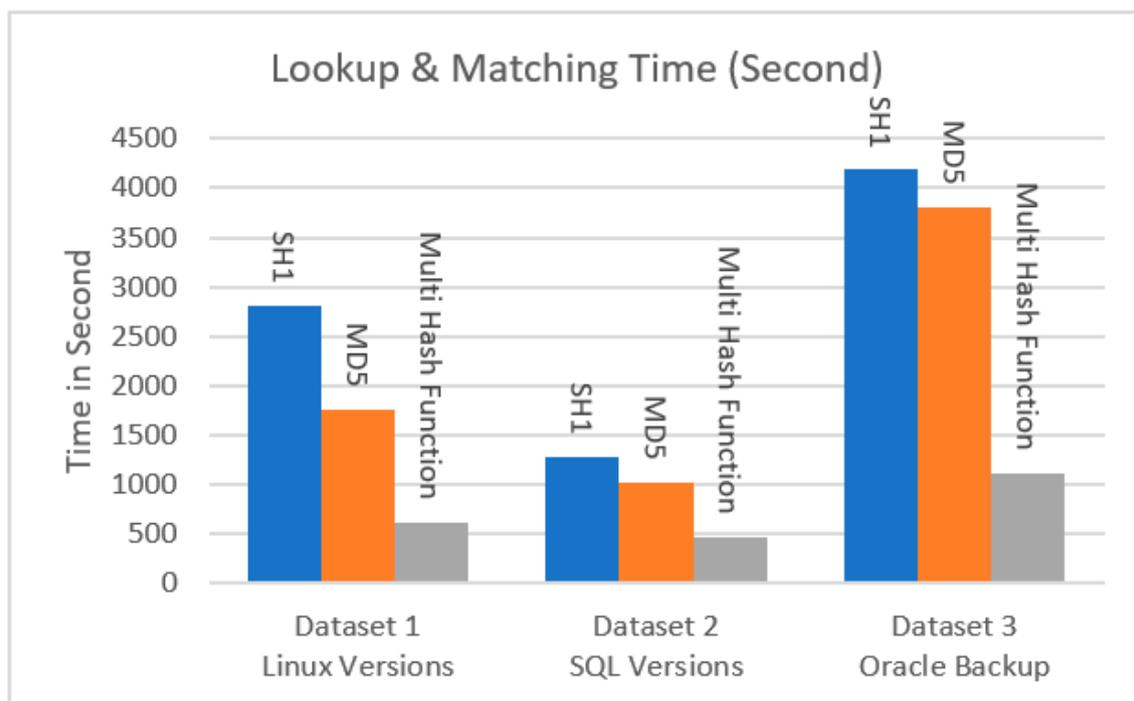
**Figure 5.** Hashing time comparison.**Figure 6.** Hashing throughput comparisons.

### 5.2.3. Lookup and Matching Time

We carried out experiments with the proposed hierarchical multi-level lookup and matching technique and the conventional full hash lookup algorithm, and we varied the hash size from 80 bytes for the proposed system, 128 bytes for MD5, and 160 bytes for SHA-1 to explore the effect of various hash sizes on the fingerprint lookup and matching time and the efficiency of the proposed fingerprint lookup algorithm. We kept track of how long each type of fingerprint lookup took. Table 5 and Figure 7 demonstrate that the total lookup time varies as the fingerprint technique is changed. As a result, one can indicate that, by utilizing our technique, fingerprint lookup is more effective and saves a significant amount of time, thus speeding up the lookup and matching stages.

**Table 5.** Lookup and matching times.

	Lookup and Matching Time (S)		
	SH1	MD5	Multi-Hash Function
Dataset 1 (Linux Versions)	2810	1765	619
Dataset 2 (SQLite Versions)	1287	1029	472
Dataset 3 (Oracle Backup)	4185	3814	1120



**Figure 7.** Lookup and matching time comparisons.

For the evaluations, we primarily used the percentage of storage saving, hashing time, hashing throughput, and lookup time as the key performance indicators; the evaluation results in Tables 6 and 7 calculated by Equations (3) and (4) illustrate the excellent foreground performance for the proposed multi-hash function compared with SHA-1 and MD5.

$$\text{Percentage of Saving} = \frac{\text{Initial Value} - \text{Final Value}}{\text{Initial Value}} * 100 \quad (3)$$

$$\text{Percentage of Throughput Enhancement} = \frac{(\text{Final Throughput} - \text{Initial Throughput})}{\text{Initial Throughput}} * 100 \quad (4)$$

**Table 6.** Multi-hash function efficiency compared to SHA-1.

	Storage Saving (%)	Hashing Time Saving (%)	Throughput Enhancement (%)	Lookup and Matching Time Saving (%)
Dataset 1 (Linux Versions)	49.97%	62.98%	171.00%	77.97%
Dataset 2 (SQLite Versions)	49.75%	48.24%	93.38%	63.32%
Dataset 3 (Oracle Backup)	50.00%	58.32%	139.04%	73.23%

**Table 7.** Multi-hash function efficiency compared to MD5.

	Storage Saving (%)	Hashing Time Saving (%)	Throughput Enhancement (%)	Lookup and Matching Time Saving (%)
Dataset 1 (Linux Versions)	37.48%	65.65%	190.00%	64.92%
Dataset 2 (SQLite Versions)	37.42%	52.09%	108.73%	54.13%
Dataset 3 (Oracle Backup)	37.50%	59.11%	143.68%	70.63%

#### 5.2.4. Hashing Collision

The collision rate is the number of occurrences in which two different data items, i.e., fingerprints, in our case, point to the same hash on the hash index table. Its proximity depends on the hash function used, although the probabilities are small, as they are nearly nonzero. However, the matter of data corruption and hash collision must be faced.

The main purpose of using the proposed multi-hash algorithm is that the second hash function reduces the number of collisions detected in the first hash function substantially, and the third hash function reduces it to zero. The collision number for Dataset 1 with the first hash function was 13,267,815; this number was decreased to 21,339 with the second hash function, and the third hash function made it zero. The fourth and fifth hash functions were utilized as further steps to prevent a collision if a chunk overpassed the first three hashes. The collision number for each hash in the system was determined as indicated in Table 8; it was discovered that five hashes are a good compromise between the size and time of the hashing index table.

**Table 8.** Number of collisions based on the number of hashes used.

	Number of Hash Collisions Detected				
	H1	H2	H3	H4	H5
Dataset 1 (Linux Versions)	13,267,815	21,339	0	0	0
Dataset 2 (SQLite Versions)	7,145,178	12,546	0	0	0
Dataset 3 (Oracle Backup)	23,549,651	37,706	0	0	0

## 6. Conclusions and Future Works

A chunk fingerprint (a hash value derived depending on the chunk's content) is a key for the hashing, lookup, and matching stages in data deduplication. The amount of chunks in a deduplication system is usually too large to keep a large (hash-table-based) index structure in RAM. Therefore, either the size of index table needs to be reduced or it needs to be stored on a secondary storage device, which reduces the system efficiency, as the load factor of an efficient hashing index table usually needs to be very low to keep the lookup time bounded.

To address these challenges, we propose a new multi-hash function based on a mathematical bounded linear hashing algorithm that promises high-performance fingerprint generation, reduces the hashing index table size, and aids in the optimization for lookup and matching fingerprints.

We evaluated the proposed system using three real datasets, and the experimental results show that it is able to obtain a significant performance enhancement in addition to storage savings. Our experimental evaluation demonstrates that the mathematical hashing

algorithm can (1) help enhance hashing time, (2) majorly reduce the hashing index table size, and (3) significantly reduce the fingerprint lookup time. We verified the efficiency of our algorithm experimentally; our results in Tables 6 and 7 show that our hashing algorithm outperforms the SHA-1 and MD5 algorithms, as it generates a mathematical chunk hash signature of 80 bytes that runs more than twice as fast as an implementation of SHA-1 and MD5. This method eliminates the needs for byte-to-byte comparisons and saves more time in the matching and lookup stage.

Future works include

- using compression for the hashing index table for additional saving,
- using a distributed hash index table (DHIT) to reduce matching and lookup I/O, and
- using a variable number of hashes based on the size of the deduplicated data instead of using five hashes.

**Author Contributions:** The model introduced in this paper was proposed by A.S.M.S., who collected the experimental data and performed the analysis. L.E.G. provided guidance for the paper and was the research advisor. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research received no external funding.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Conflicts of Interest:** The authors declare that there are no conflict of interest.

## References

1. Xia, W.; Jiang, H.; Feng, D.; Douglis, F.; Shilane, P.; Hua, Y.; Fu, M.; Zhang, Y.; Zhou, Y. A comprehensive study of the past, present, and future of data deduplication. *Proc. IEEE* **2016**, *104*, 1681–1710. [[CrossRef](#)]
2. Kaur, R.; Chana, I.; Bhattacharya, J. Data deduplication techniques for efficient cloud storage management: A systematic review. *J. Supercomput.* **2017**, *74*, 2035–2085. [[CrossRef](#)]
3. Vijayalakshmi, K.; Jayalakshmi, V. Analysis on data deduplication techniques of storage of big data in cloud. In Proceedings of the 2021 5th International Conference on Computing Methodologies and Communication (ICCMC), Erode, India, 8–10 April 2021.
4. Kim, D.; Song, S.; Choi, B.-Y. *Data Deduplication for Data Optimization for Storage and Network Systems*; Springer: Cham, Switzerland, 2017.
5. Zhang, D.; Le, J.; Mu, N.; Wu, J.; Liao, X. Secure and Efficient Data Deduplication in JointCloud Storage. *IEEE Trans. Cloud Comput.* **2021**. [[CrossRef](#)]
6. Zhu, B.; Li, K.; Patterson, R.H. Avoiding the disk bottleneck in the data domain deduplication file system. In Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08), San Jose, CA, USA, 25–29 February 2008.
7. Song, L.; Deng, Y.; Xie, J. Exploiting fingerprint prefetching to improve the performance of data deduplication. In Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing, Zhangjiajie, China, 13–15 November 2013.
8. Sujatha, G.; Raj, J.R. Digital Data Identification for Deduplication Process using Cryptographic Hashing Techniques. In Proceedings of the 2021 International Conference on Intelligent Technologies (CONIT), Hubli, India, 25–27 June 2021.
9. Lu, G.; Jin, Y.; Du, D.H. Frequency based chunking for data de-duplication. In Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Miami Beach, FL, USA, 17–19 August 2010.
10. Xia, W.; Zou, X.; Jiang, H.; Zhou, Y.; Liu, C.; Feng, D.; Hua, Y.; Hu, Y.; Zhang, Y. The Design of Fast Content-Defined Chunking for Data Deduplication Based Storage Systems. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2017–2031. [[CrossRef](#)]
11. Won, Y.; Ban, J.; Min, J.; Hur, J.; Oh, S.; Lee, J. Efficient index lookup for De-duplication backup system. In Proceedings of the 2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems, Baltimore, MD, USA, 8–10 September 2008.
12. Zhang, Y.; Wu, Y.; Yang, G. Droplet: A distributed solution of data deduplication. In Proceedings of the 2012 ACM/IEEE 13th International Conference on Grid Computing, Beijing, China, 20–23 September 2012.
13. Bhagwat, D.; Eshghi, K.; Long, D.D.; Lillibridge, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In Proceedings of the 2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, London, UK, 21–23 September 2009.
14. Lillibridge, M.; Eshghi, K.; Bhagwat, D.; Deolalikar, V.; Trezis, G.; Camble, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In Proceedings of the 7th Conference on File and Storage Technologies (FAST '09), San Francisco, CA, USA, 24–27 February 2009; pp. 111–123.

15. Lu, G.; Nam, Y.J.; Du, D.H. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In Proceedings of the 2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), Pacific Grove, CA, USA, 16–20 April 2012.
16. Kumar, N.; Rawat, R.; Jain, S. Bucket based data deduplication technique for big data storage system. In Proceedings of the 2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 7–9 September 2016.
17. Shrivastava, A.; Tiwary, A. A Big Data Deduplication Using HECC Based Encryption with Modified Hash Value in Cloud. In Proceedings of the 2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS), Madurai, India, 14–15 June 2018.
18. Li, D.; Wang, Q.; Guyot, C.; Narasimha, A.; Vucinic, D.; Bandic, Z.; Yang, Q. Hardware accelerator for similarity based data dedupe. In Proceedings of the 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), Boston, MA, USA, 6–7 August 2015.
19. Zhang, Y.; Feng, D.; Jiang, H.; Xia, W.; Fu, M.; Huang, F.; Zhou, Y. A Fast Asymmetric Extremum Content Defined Chunking Algorithm for Data Deduplication in Backup Storage Systems. *IEEE Trans. Comput.* **2016**, *66*, 199–211. [[CrossRef](#)]
20. Conde-Canencia, L.; Hamoum, B. Deduplication algorithms and models for efficient data storage. In Proceedings of the 2020 24th International Conference on Circuits, Systems, Communications and Computers (CSCC), Chania, Greece, 19–22 July 2020.
21. Kumar, R.; Lachure, J.; Doriya, R. Use of Hybrid ECC to enhance Security and Privacy with Data Deduplication. In Proceedings of the 2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC), Coimbatore, India, 4–6 August 2021.
22. Leesakul, W.; Townend, P.; Xu, J. Dynamic data deduplication in cloud storage. In Proceedings of the 2014 IEEE 8th International Symposium on Service Oriented System Engineering, Oxford, UK, 7–11 April 2014.
23. Reddy, B.T.; Kiran, P.S.; Priyanandan, T.; Chowdary, C.V.; Aditya, B.J. Block Level Data-Deduplication and Security Using Convergent Encryption to Offer Proof of Verification. In Proceedings of the 2020 4th International Conference on Trends in Electronics and Informatics (ICOEI) (48184), Tirunelveli, India, 15–17 June 2020.
24. Deepakumara, J.; Heys, H.M.; Venkatesan, R. FPGA implementation of MD5 hash algorithm. In Proceedings of the Canadian Conference on Electrical and Computer Engineering 2001. Conference Proceedings (Cat. No. 01TH8555), Toronto, ON, Canada, 13–16 May 2001.
25. Schneier, B. Cryptanalysis of MD5 and SHA: Time for a New Standard. *Computerworld*, 19 August 2004.
26. Wang, G. An efficient implementation of SHA-1 Hash function. In Proceedings of the 2006 IEEE International Conference on Electro/Information Technology, East Lansing, MI, USA, 7–10 May 2006.
27. Chan, X.; Liu, G. Discussion of one improved hash algorithm based on MD5 and SHA1. In *Proceedings of the World Congress on Engineering and Computer Science*; Citeseer: San Francisco, CA, USA, 2007.
28. Ratna AA, P.; Purnamasari, P.D.; Shaugi, A.; Salman, M. Analysis and comparison of MD5 and SHA-1 algorithm implementation in Simple-O authentication based security system. In Proceedings of the 2013 International Conference on QiR, Yogyakarta, Indonesia, 25–28 June 2013.
29. Eastlake, D.; Schiller, J.; Crocker, S. Randomness requirements for security. *RFC4086*, 2005.
30. l'Ecuyer, P. History of uniform random number generation. In Proceedings of the 2017 Winter Simulation Conference (WSC), Las Vegas, NV, USA, 3–6 December 2017.
31. Wang, L.; Dong, X.; Zhang, X.; Guo, F.; Wang, Y.; Gong, W. A Logistic Based Mathematical Model to Optimize Duplicate Elimination Ratio in Content Defined Chunking Based Big Data Storage System. *Symmetry* **2016**, *8*, 69. [[CrossRef](#)]
32. Waghmare, M.B.; Padwekar, S.V. Survey on techniques for authorized deduplication of encrypted data in cloud. In Proceedings of the 2020 International Conference on Computer Communication and Informatics (ICCCI), Coimbatore, India, 22–24 January 2020.
33. Upadhyay, A.; Balihalli, P.R.; Ivaturi, S.; Rao, S. Deduplication and compression techniques in cloud design. In Proceedings of the 2012 IEEE International Systems Conference SysCon 2012, Vancouver, BC, Canada, 19–22 March 2012.
34. Linux, The Linux Kernel Archives. Available online: <http://kernel.org/> (accessed on 1 May 2021).
35. D.RichardHipp, SQLite Kernel Archives. Available online: <https://www.sqlite.org/chronology.html> (accessed on 1 October 2020).