Summer 2021

Efficient Metadata Lookup In Inline Deduplication Systems Leveraging Block Similarity

Rakesh Gururaj
San Jose State University

Efficient Metadata Lookup In Inline Deduplication Systems Leveraging Block Similarity

A Project Report

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Rakesh Gururaj

May 2021

ABSTRACT

Data deduplication is a concept of physically storing a single instance of data by eliminating redundant copies to save the storage space. The adoption of deduplication is minimal in actively accessed primary storage because of its complexities, such as random access patterns to data and the need for quicker request response time. Most of the solutions designed for primary storage are offline and dependent on the concept of locality. This paper proposes an inline deduplication system with a Machine Learning based cache eviction policy to reduce the metadata overhead in the deduplication process, eliminate the redundant writes and improve the overall throughput in latency-sensitive storage workload. The system's major components are superblocking, categorizing superblocks, similarity detection, and deduplication supported by an efficient caching mechanism. It categorizes identical sequence of blocks based on the minimal fingerprint value of the superblock.

Caching of the fingerprints plays a vital role in improving performance during deduplication. A novel Machine Learning model for cache eviction is built based on the recency, frequency, and category of a block. The experimental results show that more than 33% of redundant writes are eliminated with smaller superblocks, the metadata overheads are minimized by at least 54.5% by categorizing similar superblocks, and the cache hit rates based on the workload-dependent Machine Learning model are higher by 5.43%,10.36% over system with LRU eviction and LFU eviction policy respectively resulting in 14.4% better throughput than a system with traditional cache eviction policy with a metadata cache allocation of 10% of average metadata stream size. The cache system learns the past evicted block I/O statistics and refines itself while choosing an eviction candidate. The system has shown satisfactory performance in all the real-world I/O traces considered for experiments.

Keywords – inline deduplication, block similarity, cache eviction, data fragmentation

ACKNOWLEDGEMENT

I want to express my deepest gratitude towards my Advisor, Mentor, and Counselor, Dr. Teng-Sheng Moh, who guided me throughout my graduate education. His enthusiasm in teaching enabled a positive environment for continuous learning and research. His persistent encouragement and support have helped me to achieve this feat. In the absence of his help, this project would not have been possible.

I would like to sincerely thank my committee member and The Chair of the department Dr. Melody Moh who kept me constantly engaged in the research and helped me connect with the Industry experts to stay up to date with the technology and advance in my research.

I express my heartfelt appreciation to my committee member Dr. Philip Shilane, Dell Technologies, and Mr. Bhimsen Bhanjois, Senior Distinguished Engineer, Dell Technologies, for their valuable feedback and direction to this research. Their insights have contributed to a greater extent in advancing this research.

I wish to express my indebtedness to my family for constantly motivating and supporting me in achieving my dream. I thank my friends and my colleagues for inspiring and helping me grow in life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

The recent buzzword in the technology industry is digitization. As part of the digital transformation, the earlier available information on paper and other sources are now transformed into digital information. With the advent of technology, there is an increase in data sources such as Internet of Things (IoT) sensors, mobile phones, personal computers, and storage systems. According to the reports from the International Data Corporation (IDC), the amount of data growth across the globe is expected to be 175 Zettabytes (ZB) by 2025 [1]. Hence, the storage of growing digital data becomes challenging. However, we cannot ascertain that all these data are unique; there is considerable duplicate data [2] that can be eliminated to incur space savings. To maintain an efficient storage system, the removal of duplicate data becomes necessary. Data deduplication becomes an essential tool in storage optimization.

Several studies [2], [3], [4] were conducted to understand the access pattern and the storage footprints of the primary and secondary storage workloads. The results suggest that deduplication can help in reducing space consumption by 10x in a backup storage environment [2]. Deduplication is not limited to space savings, but it also helps in reducing the load in the I/O path and traffic in the network. The application of deduplication is leveraged extensively in the backup storage environment. Adopting the deduplication technique in primary storage is minimal due to its operational complexities such as computationally expensive fingerprint generation, metadata overhead, and meeting performance metrics of the storage system. There are multiple primary storage systems [5] that have deduplication and compression. Compression techniques such as LZ77 and LZ78 [6], [7] find additional data within a particular data chunk and effectively reduce redundant

data. Deduplication is more efficient than the compression technique since it removes a large amount of redundant data across several files.

In the primary storage system, the access pattern and storage footprints are irregular leading to poor temporal locality [8], [9]. Therefore, it becomes a challenging task for researchers to design an efficient deduplication system with minimal impact on the performance. An efficient deduplication system for primary workload should have a high throughput characteristic with minimal latency and overhead in metadata management and duplicate elimination. Most deduplication systems efficiency focuses on metadata management [10], [11]. There exists a tradeoff between the storage gain and performance on account of deduplication. Splitting of data into smaller fixed or variable-sized chunks will increase duplicate elimination, but the overhead of storing metadata for smaller blocks increases resulting in degradation of performance.

The fundamental process involved in deduplication is chunking, fingerprint generation for chunks, and duplicate elimination. The incoming data stream is split into multiple blocks of the same or varying sizes for which a fingerprint is generated using hash functions such as MD5 and SHA1. Though the hash collision is possible with standardized algorithms, the collision rate is much lower than physical disk failure [12]. The fingerprints are then compared with fingerprints in the disk, and the duplicates are eliminated. When a block is identified as a duplicate block, the system will not store the duplicate block; however, a reference to the unique block will be stored in place of the duplicate block, and the reference count of the unique block will be increased. Blocks are flagged for garbage collection to reclaim the space if the block is no longer referenced by any file.

The process of deduplication can also lead to disk fragmentation issues. After deduplicating the incoming data, the block sequence in a file can be scattered across the

disk. The performance of read request in these scenarios will degrade because multiple I/O operations have to be made to collect and organize the sequential data. The above challenges are addressed by building an inline deduplication system with Machine Learning based cache eviction. The primary focus of this design is to reduce the metadata overhead, eliminate the redundant writes, and improve the overall throughput in latency-sensitive storage workload while performing deduplication. The results of the experiments shows that the system works well with an environment exhibiting poor locality information and random-access pattern. The core concept of the system is to break down the incoming stream of data into superblocks and categorize similar superblocks sharing the same metadata, thereby limiting the metadata lookup of those superblocks only to a particular category. The categorization of the superblock is based on the minimal fingerprint of the superblock. The system leverages similarity within superblocks to reduce the metadata overhead and to improve the performance while deduplicating. The entire similarity detection process is supported by a Machine Learning based cache eviction policy. The significant contributions from the project are as follows.

1) Building an inline deduplication system for reducing the metadata overhead, eliminating duplicate requests, increasing the system throughput, and reducing the latency.
2) An algorithm to split the incoming data stream into superblocks and categorize similar superblocks based on the minimal fingerprint of superblock to confine the metadata lookup space for duplicate identification and elimination.
3) Developing a novel workload-dependent Machine Learning model for cache eviction based on the recency, frequency, and category of a block to increase the cache hit rate thereby increasing the throughput of the system.

4) Using Bloom filter data structure to reduce the disk lookup while generating a new category.

The rest of the paper is organized as follows: Section II provides the necessary background of deduplication process. Section III describes existing deduplication for primary storage systems. Section IV describes the design, flow, architecture, and implementation of the deduplication system. Section V describes the results obtained from various experiments. Section VI discusses the conclusion and describes the future scope of this project.

II. BACKGROUND

Deduplication as a process is dependent on multiple parameters. We can obtain maximum gain by tuning the deduplication parameters. The categorization of the deduplication process can be as follows.

1) Size based deduplication
2) Time based deduplication
3) Node based deduplication
4) Environment based deduplication

Various categories and sub-categories of deduplication system are shown in Fig. 1.

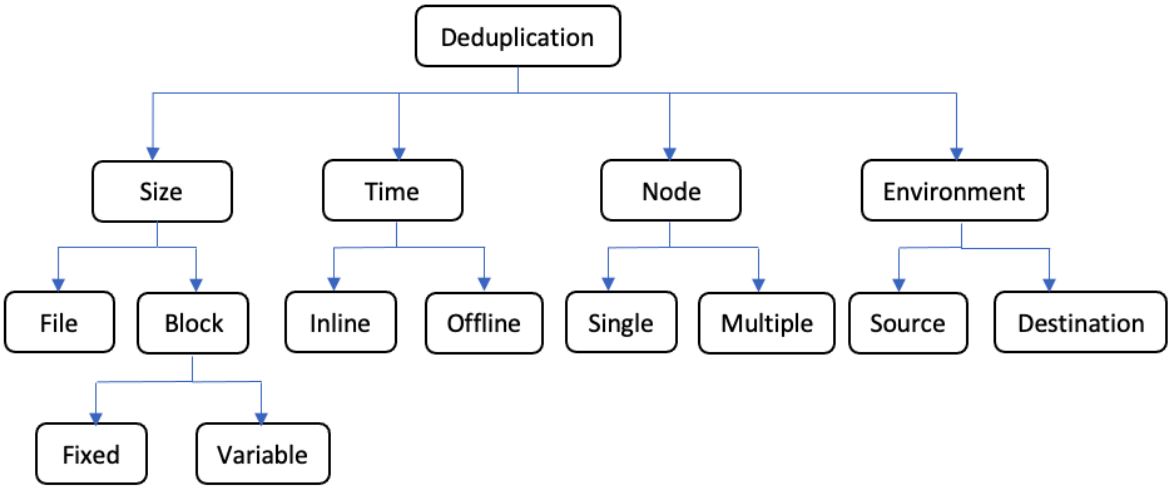

Fig. 1. Categories of deduplication

A. Size Based Deduplication

The basic unit of deduplication can depend based on the choice and the overhead of duplicate elimination.

1) *File-level:* The basic unit for deduplicating is an entire file. Each file is considered as a single unit for duplicate detection and elimination. During the process of deduplication, the fingerprints are calculated for the entire file. This technique can flag a duplicate only if two files are identical and contain the exact same content. The metadata overhead in this type

of deduplication is relatively lower because fewer chunks are created while deduplicating. The limitation of this technique is that the deduplication ratio is poor because only identical files are considered duplicates. However, it is much simpler to implement this technique when compared to other deduplication techniques.

2) *Block-level:* As mentioned in the earlier section, the incoming data request is split into several blocks. The basic unit of deduplication is a block. These blocks are later processed for duplicate detection. We can further categorize it into two types based on the size of the block.

a) *Fixed-size:* In this approach, we define the block size such as 4 KB, 8 KB, 16 KB, 32 KB, 64 KB blocks, and so on. Based on the choice of the size, the incoming data stream is split uniformly into fixed-size blocks; even with the fixed-size blocking, the last block of a file is either a small block or it is zero-padded. Ideally, fixed-size blocks are more superficial and straightforward to implement. However, the deduplication ratio will be decreased since any minor change to the file's content will change the boundaries of the blocks.

b) *Variable-size:* This approach uses a Content Defined Chunking (CDC) algorithm to split the data stream into variable-length blocks. When the block contents are changed or modified, the border of the blocks is altered accordingly and split. It gives a higher deduplication ratio when compared to fixed-size chunking because of handling the contents effectively. Since each block can have varying sizes, the management of the blocks is complex at the storage level resulting in higher metadata overhead. CDC is computationally expensive while finding the exact border for splitting.

B. Time Based Deduplication

The appropriate time for carrying out the deduplication will depend on the storage requirement such as performance, I/O traffic, the number of disk writes.

1) *Inline deduplication:* The deduplication process is triggered as there is an incoming data stream. It splits the incoming stream, generates the fingerprints for the blocks, and eliminates duplicates before storing the data block on the disk. The need for storing the data in a temporary location is avoided. The main drawback of this approach is having overhead in the I/O path while performing deduplication. The response of the system might be degraded.

2) *Offline deduplication:* This technique does not deduplicate the data in real-time. When the system is idle, the deduplication process is triggered. However, this technique requires a temporary location to store the data, but the performance of the critical path remains unaffected. The deduplication is triggered post processing of incoming data. It is difficult to identify the right time for triggering the deduplication process.

3) *Hybrid deduplication:* This technique combines both the inline and offline deduplication techniques to achieve a higher deduplication ratio. The primary focus of this technique is to achieve maximum deduplication benefit through inline deduplication. However, a threshold for latency is maintained. If the latency increases beyond the threshold, the data is stored on the disk and deduplicated in an offline manner. The complexity in this technique lies in managing the file data that can be in two different states.

C. Node Based Deduplication

Deduplication can be applied to each node or a cluster of nodes based on the requirement and architecture of the storage system.

1) *Single node:* Each node in the storage system will be installed with an independent deduplication engine. Only the data that is inside that node will be considered for duplicate comparison and elimination during the deduplication process. It is easier to manage the

metadata for each node separately. However, the deduplication ratio will be higher only within the node, but it fails to identify the identical blocks existing in other nodes resulting in a lower deduplication ratio across the cluster.

2) Multi-node: A centralized or distributed metadata server can be chosen to store the metadata obtained from all the nodes. Deduplication can be applied to all the nodes across the cluster. The deduplication ratio across the cluster is higher than single node since the duplicate blocks can be identified even if the block is scattered across cluster nodes. However, there is an additional overhead to maintain a central metadata server. The operational cost of detecting a duplicate in a centralized server is higher than the cost involved in a single node [2]. It also depends on the decision of the sending the data to the right node for deduplication. If the data is sent to a wrong node, the deduplication ratio can get lower than the benefit achieved through single node system.

D. Environment Based Deduplication

In a backup or archival environment, deduplication can be carried out in either the source system or at the destination system based on the need and availability of the resources.

1) Source: Deduplication is performed in the data origin environment. The deduplicated data is sent over the network to the destination system. This approach helps in reducing the number of packets that are sent and thus require lower bandwidth. Employing this technique in low bandwidth and low space-constrained destination systems is beneficial. But the computation of the source environment should be higher.

2) Destination: The non-deduplicated data is transferred from the source environment to a backup or archival environment. Deduplication is triggered after the receipt of the data at the destination environment. This reduces the load in the source system. But it requires

higher bandwidth for transferring data from source to destination. We can leverage the idle time of the system to perform deduplication.

E. Cache Eviction

The latency incurred in serving a request in a latency-sensitive primary storage system should be minimal. It is important to reduce the overall latency in a system by maintaining the hot data in the faster accessed medium and evicting the colder or less accessed data from a faster access medium to slower access medium. Caching plays a vital role in maintaining the hot data during the deduplication process. Primary storage workloads exhibit random access patterns, making it difficult to choose an ideal cache eviction strategy. The recency, frequency, current sequence, future sequence of a block are the most common parameters which decide the eviction candidate when the cache is full. Designing an algorithm that can yield hit rates closer to an optimal caching algorithm is significant. In our current system, each element in the cache represents a block structure. Each block structure contains a fingerprint, Physical Block Address (PBA) and reference count of that particular block.

1) *BELADY'S Lookahead Page Replacement:* This algorithm provides an optimal hit rate for a direct-mapped cache and serves as a baseline for evaluating the other cache eviction strategies [13]. It helps in directing the nonlookahead algorithms to increase the gain as closer to optimal gain. It considers the future usage statistics of blocks such as the recency and/or frequency to choose an eviction candidate that can be used furthest in the future. This is a theoretical approach that provides an optimal hit rate with a minimal cache miss rate. However, this algorithm cannot be implemented in a real-world situation since we cannot predict the blocks that will be accessed in the future.

2) *Least Frequently Used (LFU):* A counter is placed to record the number of times a

block is used in the cache. It records the frequency of a block based on the number of times the block is requested while in the cache. Once the block is evicted from the cache, the counter is set to 0. It depends on the logic that a block that has been accessed frequently has higher chances that it will be accessed in the future block accesses. However, the less frequently accessed blocks in the cache are chosen as eviction candidates and evicted from the cache.

3) *Least Recently Used (LRU):* The LRU will keep recording the block recency. When a block is accessed, it is removed and added to the queue to maintain the recency. The blocks that are least recently accessed are evicted from the cache during an eviction. It depends on the logic that a recently used block has higher chances to be accessed in the future block accesses.

F. Machine Learning (ML) Algorithms

The statistics of recency and frequency of a block can build an ML model to find out the suitable candidates for eviction when the cache is full. We have used two supervised ML models that are trained with the incoming data stream.

1) *Random Forest Classifier (RFC):* It is an ensemble approach with multiple decision trees working together towards generating a classification [14]. Each decision tree will generate a class, and the class that is voted by most of the decision trees is considered the model's output. A setup with an ensemble will yield better results than an individual tree model. The classifier model will take the incoming block requests as an input, analyzes the statistics of blocks that are evicted in the past and generate the eviction candidates based on the current items in the cache. It classifies the noneviction candidate as class 0 and the eviction candidates as class 1.

2) *K-Nearest Neighbors Classifier (KNNC):* It is an approach where the nearest items

are similar with higher probability. In other words, similar items will be closer to each other. Therefore, it is important to find and associate the elements that are in closest proximity to a given data block entry in cache. The distance between the data blocks is measured using the Euclidean distance. Based on the distance between the two items, we can measure the similarity. The rationale behind using a KNNC is to explore the similarity between the blocks that are present in the cache and associate similar blocks for the incoming cache entry. Depending on the value of "K (number of neighbors)," the items will be aligned into multiple groups. The model will associate all the similar elements and retain similar elements in the cache while selecting the elements that are not similar as eviction candidates. It classifies the noneviction candidate as class 0 and the eviction candidates as class 1.

3) Scikit-learn: It is a library that provides various ML algorithms [15]. An ML algorithm from scikit-learn can pre-process, fit, and generate new user data based on the need. It provides support for both supervised and unsupervised ML algorithms. NumPy is a scientific library that helps perform analysis with the data. We have leveraged the Scikit-learn library for implementing the RFC and KNNC algorithm.

III. RELATED WORK

Disk bottleneck and performance degradation are the two critical issues in implementing a primary workload. Deduplication has seen its success for backup and archival environment [2], [18-21]. Most of the solutions for the primary storage are implemented as offline mode. Few of the productized offline storage systems are EMC Celerra [5] and NetApp ASIS [22]. Locality and similarity are the two concepts that have been explored extensively in the primary storage system while implementing deduplication [11], [17], [23], [24].

iDedup [17] is considered the pioneer in the inline deduplication system implemented for the primary storage workload. It exploits spatial locality to store data on the disk and supports sequential access and temporal locality to build an effective cache system. Unfortunately, the primary workload does not exhibit extensive locality property. iDedup ignores the smaller files and requests that are below a threshold value to improve the performance. It is also dependent on the underlying file system. The Partially Dedupped File System (PDFS) [8] segments the incoming data stream and applies Locality Sensitive Hashing (LSH) to find similar blocks. The LSH technique is complex and computationally expensive. Therefore, it is challenging to implement PDFS for the real-time primary workload. Performance Oriented Deduplication (POD) [23] focuses primarily on minimizing the performance degradation while deduplicating. POD assumes a temporal locality in the primary workload and indexes the fingerprint and metadata based on the locality resulting in improved performance in the I/O path during deduplication. It is difficult to witness performance gain in workloads exhibiting poor temporal locality. Heuristically Arranged Non-Backup Deduplication System (HANDS) [24] is another approach exploiting the temporal and spatial locality of the data stream. It employs several heuristic methods to

index the fingerprint to reduce the number of lookups and increase the system's overall performance. Nevertheless, it fails to address the issue of random-access patterns in primary workload.

Hybrid Deduplication Systems (HDS) is efficient in deduplicating the data since it involves both inline and offline deduplication. Steam Locality Aware Deduplication (SLADE) [27] assumes temporal locality in the data stream and designs a cache of fingerprints based on the temporal locality. On the other HDS – A Block-Level Similarity-Based Approach [10] exploits similarity between the data segments and uses the locality preserving indexing built in the form of a graph to improve the performance on the I/O path. However, using the graph data structure to preserve the locality is an expensive operation in any modification to the structure.

IV. DESIGN AND IMPLEMENTATION

The primary objective of this research is to reduce the metadata overhead and response time while achieving higher deduplication ratio, increased overall throughput and higher cache hit rate. The capacity optimization is driven by the large files, whereas performance optimization is driven by smaller files. The incoming data stream is divided into multiple superblocks of fixed size that can be configured depending on the user requirement. Each of the superblocks is further divided into fixed-size blocks. The blocks are sent to a fingerprint generator that hashes the data using hashing algorithms such as MD5 and SHA-256. The minimal fingerprint of the superblock is calculated and considered as the Super Block Representative (SBR). The categorization of similar superblocks is based on the SBR value. Each block's fingerprints are compared with other existing fingerprints within the same category. The performance of the deduplication engine is dependent on the caching strategy. Workload-dependent ML model is used to evict the items in cache while targeting to achieve cache hit rate near to optimal hit rate described in previous section. Categorization of similar superblocks into the same category and effective cache management serves as the backbone to the system resulting in a reduction of lookups and disk I/O operations during deduplication of data. Bloom filters are used to check if an SBR exists in the system and helps in category management. The overall architecture of the system is shown in Fig. 2.

A. Terminologies

Before describing the system's design, it is essential to understand the terminologies used as part of the prototype.

1) *Superblock:* Superblocks are the basic unit for file organization in deduplication

enabled storage. The size of the superblock is fixed and configurable. However, for the experimentation of the system, multiple superblock sizes are used, such as 32 KB, 64 KB, 128 KB, 256 KB of size. A larger file is split into multiple smaller superblocks based on the size of the file. The illustration of a file that is split into superblocks is depicted in Fig. 3.

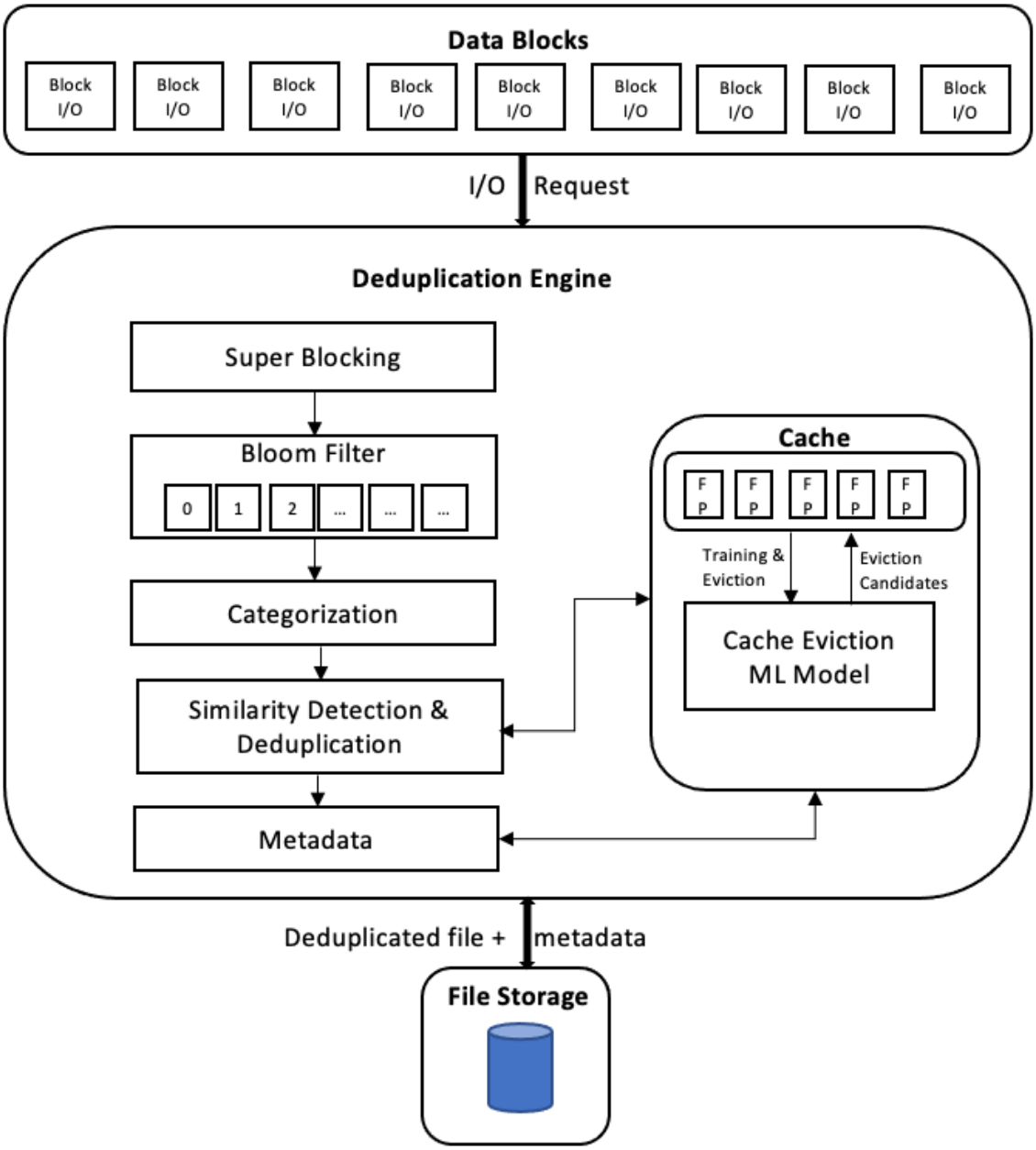

Fig. 2. Architecture of deduplication system

2) *Block:* Each of the superblocks is further divided into multiple blocks of fixed

length. Variable-sized blocks can be used in the backup environment where the size of the files is often larger than 100 MB. An insertion or deletion can shift the boundary of the block. Therefore, using variable-sized blocks is beneficial in this setup. However, in the primary storage system, each file is lesser than 1 MB, and hence, using a fixed-sized blocking will reduce the latency incurred by variable-sized blocking while achieving most of the potential deduplication. A block is the smallest unit for deduplication. We use the block size of 4 KB for the experiments.

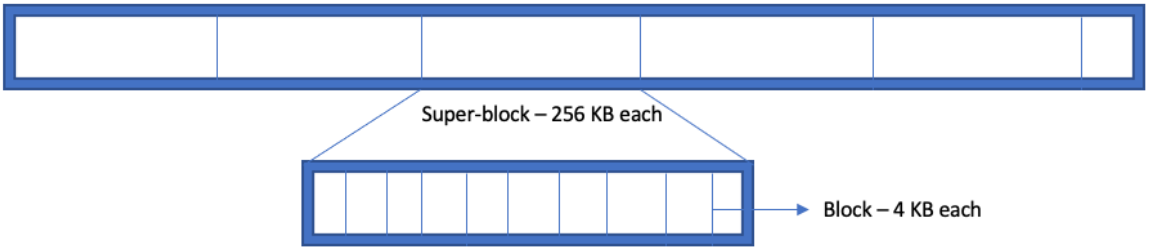

Fig. 3. Representation of superblock and block

3) Category: For each of the superblocks, the smallest fingerprint value of the block is considered as SBR and it is passed through a Bloom filter to determine if the SBR is present in the system. If SBR does not exist, a new category is created for the superblock and blocks within the superblock is added to the newly created category. When an identical SBR is found, the incoming superblock is categorized into the same category as SBR and only unique blocks are added to the existing category. Superblocks belonging to the same category share metadata information. The existing system does not have a limit on number of unique blocks for a category.

4) Deduplicated file layout: The deduplicated file contains the superblocks and blocks. Duplicates that are part of the deduplication process contain references to the unique blocks. The illustration of the layout of the deduplicated file is depicted in Fig. 4. As

described in the figure, each of the deduplicated files has a category ID and the fingerprint reference where a duplicate is replaced with a reference.

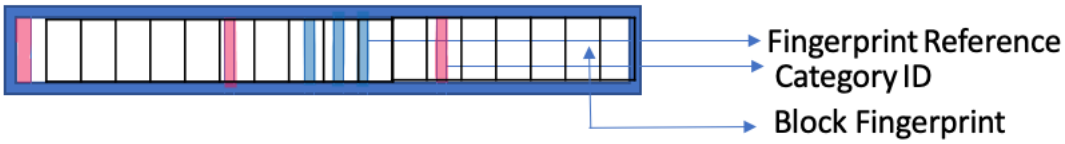

Fig. 4. Representation of deduplicated file

B. Data Structure

Data structures are essential in metadata management. Several different data structures must be maintained for each block of data. For each of the superblock, an SBR must be mapped with a category ID. Blocks within a superblock should be mapped with the category ID, fingerprint, and a counter for each block reference. Logical Block Address (LBA) to Physical Block Address (PBA) mapping must be established. Similarly, for each of the PBA, a category ID must be mapped. The current system considers the smaller files that are of size lesser than the superblock. A hash table of fingerprints of each block is maintained to address small files. Fig. 5. — Fig. 8. illustrates the various data structures that are used as part of the prototype.

SBR	Category ID
---	---

Block Fingerprint	Physical Block Address (PBA)	Count
---	---	---
---	---	---
---	---	---

Fig. 5. SBR to category ID mapping and category ID to blocks mapping

Block Fingerprint	Physical Block Address (PBA)	Count
---	---	---
---	---	---
---	---	---

Fig. 6. Hash table for smaller files

LBA	PBA
---	---
---	---
---	---

Fig. 7. LBA to PBA mapping

PBA	Category ID
---	---
---	---
---	---

Fig. 8. PBA to category ID mapping

C. Similarity Detection

The key idea of the system is to categorize similar superblocks into the same category. Metadata of the blocks in the same category such as fingerprints, count of all the blocks inside a superblock, LBA to PBA mappings are stored in individual categories. To find the similarity between different superblocks, we leverage the concept of Broder's theorem [28]. According to the theorem, superblocks are similar to each other, with a higher probability if the smallest fingerprint of the superblocks is similar. When the superblocks are similar, it shares most of the underlying block between them. For illustration, consider two superblocks SB1 and SB2. Let FP1 be the smallest fingerprint of SB1 and FP2 be the smallest fingerprint of SB2. If FP1 and FP2 are the same, then SB1 and SB2 have a higher probability of being similar. During categorization of the superblocks, we consider the smallest fingerprint of the block as SBR and represent the superblock with SBR. A decrease in the size of the superblock will help in identifying more similarities between the files. The Broder's theorem can be summarized as the below equation.

$$Probability(FP1 = FP2) = \frac{|SB1 \cap SB2|}{|SB1 \cup SB2|} \quad (1)$$

When similar superblocks are categorized, the deduplication system will group all the blocks within the superblocks into the existing category. If the superblocks are not similar, a new category will be created, and the blocks of the superblock will be added to

the new category. Identifying identical blocks in the incoming data stream and categorizing them into the same bin helps exploit block similarity, reducing the overhead incurred in metadata lookup. This approach limits the duplicate comparison and elimination to the blocks within a particular category ID.

D. Bloom Filter Implementation

A probabilistic data structure that can reveal if an element is present in the set or not [29]. It is hugely memory efficient and provides the output rapidly since it does not store the actual data within the data structure. The Bloom filter can certain if an element is certainly not present in the set of values. They can produce false-positive results, and therefore it will always reveal if an element might be on the set or not. The Bloom filter cannot produce a false-negative result. However, we can control the false-positive rate by varying the parameters of Bloom filter such as, increasing the Bloom filter's size and using different number of hash functions. We have leveraged the Bloom filter to find if the SBR already exists in the metadata. The SBR that does not pass through the Bloom filter is the candidate for a new category. The workflow of the Bloom filter is depicted in Fig. 9.

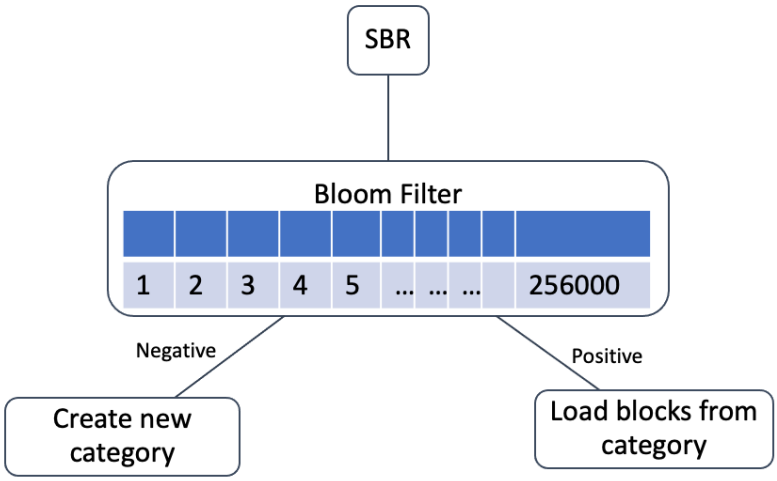

Fig. 9. Bloom filter implementation

We have used the Bloom filter package provided by the Guava library [25]. It is important to vary the Bloom filter size based on the size of the incoming data stream. Therefore, it is critical to choose the expected number of elements that might be entered into the Bloom filter beforehand. In this project setup, the size of the Bloom filter has two dependent parameters – number of block I/O and size of the superblock. Based on this, we can obtain the expected number of elements for a dataset. False-positive rates of the Bloom filter can be decreased by increasing the size of the Bloom filter. Since the Bloom filters are memory efficient and occupy minimal space, we can consider having the filter in the memory. It is important to note that Bloom filters do not store the actual data; it simply verifies and returns if an element exists in a set.

E. *Request Processing*

The request to a file comes as a data stream containing LBA's of the read or write operation. All the incoming write requests are split into multiple superblocks. For each of the blocks in the superblock, a fingerprint has been generated. Based on the fingerprints obtained for each superblock, an SBR is identified representing a superblock. Each SBR is checked for its existence in the storage. If an SBR is already present, then the respective superblock might contain identical or duplicate blocks based on the similarity detection algorithm mentioned in the earlier section. If an SBR is not found, a new category is created along with the respective metadata into the storage. The cache is updated with the newer category and superblock details.

F. Prototype Workflow

The primary workflows that are part of the system are the write request to files of size larger than or equal to the size of a superblock, smaller files that are lesser than the

superblock's size and read requests to the files. Each of the requests is handled and processed separately. After identifying the SBR for a superblock, deduplication is applied to the superblock. Typically, the number of smaller requests is greater than large requests in the primary storage workload resulting in greater number of duplicate requests [3], [4]. These smaller requests are ignored by most of the deduplication systems [17] because of the overhead involved in finding the duplicates in the smaller requests. However, as mentioned earlier, identifying duplicates in smaller requests can eliminate duplicate I/O requests resulting in enhancing the system's performance. In this project, smaller requests are handled effectively to avoid duplicate write requests. The metadata information for the smaller request is stored separately with category ID as 0. It becomes convenient to look up the hash table while processing the smaller requests. The size of the category ID as 0 will grow depending on the number of small requests that are served in primary storage.

The maximum space savings can be obtained from the large requests. This project focuses on deduplication at the superblock level. Performing deduplication at a block-level will increase in deduplication ratio and save more space. However, this approach will lead to disk fragmentation resulting in multiple metadata access while performing a sequential read of deduplicated file. On the other hand, when we deduplicate at the superblock level by maintaining a threshold of matching blocks to deduplicate, the consequences of disk fragmentation can be avoided. Deduplication in our setup is applied when two blocks belong to the same category, and the number of identical blocks is greater than or equal to the threshold value defined by the user. When the blocks do not satisfy the above condition, the block inside the superblock is not deduplicated even if the superblocks contains duplicate blocks, and they are added to the metadata of the existing category. The

algorithms Algorithm 1, Algorithm 2, Algorithm 3 describe the process flow of various requests. Notations used in the algorithms are shown in Table I.

TABLE I
ALGORITHM NOTATIONS

Definition	Notation
Incoming Blocks, Blocks from category	B, B_C
Superblock consisting of blocks	SB
Block Fingerprint	FP
Category	C
Request I/O	R
Super Block Representative	SBR
Bloom Filter	BF

Algorithm 1: Handling write requests of file size larger than superblocks

1. Split the incoming request R into set {SB}
2. **foreach** SB in set {SB} **do:**
3. Split into further Fixed-size B and generate set {B}
4. **foreach** B in set {B} within a SB **do:**
5. Calculate FP by MD5 (B)
6. Add FP to set {FP}
7. **end for**
8. SBR = min (set {FP})
9. Pass SBR through BF and get C
10. **if** SBR does not pass:

11. C = Call **CreateNewCategory (SBR)**
12. Call **InsertIntoMetadataTables** (SBR, B, C)
13. **else:**
14. set {B_C} = **Call LoadCategoryBlocks (C)**
15. **foreach** B in set {B} **do:**
16. Check if B is in set {B_C}
17. **if** exists **then**
18. Replace B with reference to unique block
19. unique block count += 1
20. **else:** Write B to set {B_C} and load B in C
22. Call **UpdateMetadataTables (SBR, B_C, C)**
23. **end if**

24: Write Unique blocks back to storage and **end for**

Algorithm 2: Handling write requests of smaller files

1. **foreach** B in set {B} of R **do:**
2. Calculate FP by MD5 (B)
3. Add FP to set {FP}
4. B_C = **Call LoadBlockSmallRequestMetadata (B)**
5. **if** exists **then**
6. Replace B with reference to B_C
7. unique block count (B_C) += 1
8. **else:**
9. Call **InsertBlockSmallRequestMetadata (B)**

10. **end if**

11: Write Unique blocks back to storage

12: **end for**

Algorithm 3: Handling Read request

1: Read the request R

2. Lookup LBA to PBA mapping table in cache

3. **If** exists **then:**

4. Load the block data

5. Add block to File construction buffer by resolving block references

6. **else:**

7. Fetch B from the disk

8. Add block to File construction buffer by resolving block references

9. **end if**

10. Return constructed file

Each of the requests is served back to the user as per the algorithm mentioned above. The deduplication is performed inline where the incoming data stream deduplicates the block I/O and stores only the unique blocks into the storage. The metadata of each block plays a vital role in deduplicating the file. Multiple procedures are called within the algorithms mentioned above. The description of the procedure calls Proc 1 - Proc 6 are explained below. Care has been taken while deduplicating so that the performance of the system does not degrade.

Proc 1: CreateNewCategory (SBR)

1: Insert SBR into C — SBR table

2: return C

3: **If** C is not null **then**:

4: Insert SBR into BF

5: **else:** Notify failure to user

6: **end if**

Proc 2: InsertIntoMetadataTables (SBR, B, C)

1: Insert set {LBA (B)} in LBA — PBA table

2: return PBA of LBA

3: Insert set {B} with PBA (set {B}), C into Category Metadata table

4: **If** PBA exists **then**:

5: Increase the count of PBA by 1

6: **else:**

7: Set PBA count of FP to be 1

8**: end if**

9: Insert into PBA — C table

10: return Acknowledgement

11. Call CachingProcedure (C)

Proc 3: LoadCategoryBlocks (C)

1: Fetch set {B} from Category Metadata table for C

2: return set {B}

3: Call CachingProcedure (C)

Proc 4: UpdateMetadataTables (SBR, B_C, C)

1: Update / Insert set {LBA (B_C)} in LBA — PBA table

2: Update / Insert set {B_C} with PBA (set {B_C}), count (PBA), C into Category Metadata table

3: Update / Insert into PBA — C table

4: return Acknowledgement

5: Call CachingProcedure (C)

Proc 5: InsertBlockSmallRequestMetadata (B)

1: Update / Insert LBA (B) in LBA — PBA table

2: return PBA of LBA

3: Insert B with PBA (B), 0 as C into Small Request Metadata table

4: **If** PBA exists **then**:

5: Increase the count of PBA by 1

6: **else:**

7: Set PBA count of FP to be 1

8**: end if**

9: Insert into PBA — C table

10: return Acknowledgement

11. Call CachingProcedure (C)

Proc 6: LoadBlockSmallRequestMetadata (B)

1: Fetch set {B} from Small Request Metadata table for C

2: return set {B}

3: Call CachingProcedure (C)

The above procedure and algorithm perform inline deduplication of incoming data stream. The categorization of superblocks restricts the number of block comparisons for duplicate comparisons with the blocks in the same category.

G. Caching

Deduplication is a computationally expensive process. The system's performance while deduplicating the data depends on how quickly we can access data and metadata for comparing the duplicates. Caching of data and metadata plays a significant role while deduplicating. It is relatively difficult to build a cache due to random access patterns and poor locality in primary workloads. However, the block's recency – absolute last access time and frequency – number of times a block is accessed while the block is in cache can help to build a cache system that can yield higher hit rates. The success of cache management depends on two factors — building an effective cache eviction strategy and prefetching of blocks based on heuristics. An efficient cache will reduce the number of metadata lookups to the disk. The data structures described above are used for caching the metadata information. The system requires minimal cache size for storing and processing the fingerprint and other blocks related information. The current system uses the derived statistics of a block – recency, frequency, category (4 byte), and LBA (4 Byte).

1) *Data collection:* Belady's Lookahead replacement algorithm is used to find the block that can be the eviction candidates. We have two parameters for collecting the data to implement an ML based cache eviction strategy — Sampling Frequency and Eviction count. Sampling Frequency is a value for sampling the data in the cache and collecting the statistics of blocks present in the cache. These statistics are the features of a block. Eviction count provides us the number of eviction candidates that can be generated for eviction when the cache is full. We pass the data stream to Belady's algorithm and find out the

blocks that can be evicted at that instance. Therefore, at every sampling frequency instance, we collect the recency, frequency, category, and LBA of a block and their eviction status (0 for No eviction, 1 for Eviction) as per the number of candidates given in the eviction count. Based on both the parameters, several experiments have been conducted, and the results are presented in the next section. This data obtained is used as an input to the ML model that will be built.

2) Normalization: Each of the features considered above has different covariances, resulting in distortion in the data. The value of each feature is normalized to a value between 0 and 1 to overcome the above constraint. All the feature values are normalized and later sent as an input to the ML model

3) Data for ML model: As mentioned earlier, the features of an entry in the cache are recency, frequency, category, and LBA. The features are selected after feature engineering each property of a block. The following section will depict the result of the feature engineering. Below Fig. 10. depicts the data input that is given to a supervised ML model.

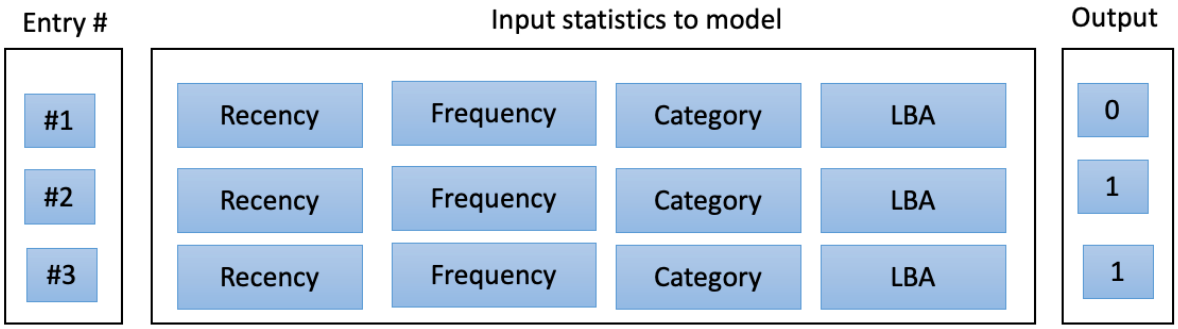

Fig. 10. Input for ML model

4) Training ML model: Several experiments were performed with three hyperparameters – sampling frequency count, eviction count, and cache size. For each of the experiments, hit rate has been calculated and the parameters yielding highest hit rate

for the ML model are considered for the dataset. The ML model is trained with the dataset generated described in above section. Sampling the cache and retrieving the statistics of the entries in the cache based on the sampling frequency has helped us avoid overfitting the model. Several blocks in the dataset that have been accessed more frequently and are not evicted for a more extended period. Similarly, the eviction count helps in choosing the candidates with highest probability of getting evicted.

We developed two different ML models using scikit-learn library and evaluated each model's performance based on the hit rate and the time consumed to generate eviction candidates.

- KNNC: The model has been trained with the dataset by varying the hyper-parameter — number of neighbors. The training time for KNN is faster when compared to RF model. The number of features to this model remained the same as described in the earlier section.
- RFC: The training data is made to fit by varying multiple hyper-parameters —max depth of decision tree, minimum sample split for each tree, number of trees in the forest). The training time for the RF model is relatively higher than the KNN model since it involves results from multiple decision trees.

At the end of each day, the data blocks that have been served so far will be sent to Belady's algorithm to generate the input statistics to the model. The ML model gets trained on the statistics, and when the cache is full, the model generates the eviction candidates based on the data present in the cache instance at that period.

5) K-fold Cross Validation: The primary storage workload witnesses random access to the data blocks. It is crucial for the model to get trained on a dataset that represents the overall data of the workload. Cross validation is employed to fit the training data well and

increase the accuracy of the model. This project uses a 5-fold cross validation strategy to train and validate the model's performance. We shuffle the input data and split them into five groups. Each time, four parts of the data are sent for training the model and one part for the validation.

6) Testing: Apart from validation, the actual testing is done when the cache is full. Each of the blocks in the data stream is captured in the cache, and when the cache is full, the ML based eviction algorithm as described in Algorithm 4 is called. The algorithm takes the current cache instance statistics and predicts the eviction candidates based on the probability of eviction. The candidates that are predicted by the ML algorithm are evicted and replaced with the incoming data block. The ML model's hit rates are analyzed and the model yielding the highest hit rate is deployed as a cache eviction strategy.

Algorithm 4: CachingProcedure (C)

```
1: foreach B in SB:
2:     if B in cache then:
3:         hit += 1
4:         update recency and frequency statistics
5:     else if cache not full then:
6:         Add B to cache
7:         update recency and frequency statistics
8:         miss +=1
9:     else if cache is full then:
10.        evict_candidates = MLmodel (C)
11:        replace evict_candidate [0] with B
12:        update recency and frequency statistics
```

13: miss += 1

14: **end if**

15: **end for**

The system has two independent modules where one of the modules takes care of superblocking and deduplicating and the other module consists of logic for ML based eviction model. Both the modules are integrated using API. We have used a third-party library Jython [26], to establish communication between both modules.

V. EXPERIMENT AND RESULTS

The system was built on Linux Operating System (OS) running on a 2.2 GHz Quad-core Intel i7 processor with RAM specification — 16 GB 1600 MHz DDR3 memory. All the experiments have been conducted in the same setup with different datasets. The objective of conducting multiple experiments is to find the correct value for each hyper-parameter used in the system and show that the system can handle a primary storage workload with lower metadata overhead.

A. Dataset

We use a publicly available data source. It consists of I/O block traces collected from the three production systems and available as FIU block trace [16]. The I/O details were recorded from Virtual Machines (VM) hosting a web server, Computer Science department email server, and a file server dedicated to researchers. The I/O traces were collected for 21 days using blktrace – mechanism to trace blocks. The details of dataset are provided in Table II. Each of the records in the I/O trace file consists of the following

- Timestamp
- Process ID
- Process name
- LBA
- Size allocated in 512 bytes
- Request type – Write or Read
- Major device number
- Minor device number
- Fingerprint – MD5 per 512 bytes

TABLE II
DATASET STATISTICS

	Homes (approx.)	**Web Server (approx.)**
Total number of requests	17.83 Million	14.29 Million
Total number of Read I/O	0.72 Million	3.11 Million
Total number of Write I/O	17.11 Million	11.17 Million

For experimentation, a low memory cache has been used to store the fingerprints of the block for faster duplicate comparison. Typically, a 10% of average everyday working data stream size is allocated for cache. However, we have experimented with multiple cache sizes to understand the performance of the system. We have considered 21 days of I/O traces from 2 production systems during the analysis of the system.

B. Feature Engineering

An ML based cache eviction model is built and integrated as part of the deduplication engine. It is important to understand the features of the model. As mentioned in the previous section, there are nine features representing block I/O requests. However, we can derive other features from the given block I/O tracer file. We have considered three derived features from the dataset. The importance of each of the features is shown in Table III. To understand the importance of each feature, we have leveraged Recursive Feature Elimination from scikit-learn library. Each of the features is ranked and the top 4 features are considered for building the model.

TABLE III
FEATURE ENGINEERING

Feature	Rank	Score
LBA	1	0.31046
Category ID	1	0.22911
Frequency	1	0.21896
Recency	1	0.2145
Fingerprint	2	0.01991
Timestamp	4	0.00283
Process ID	3	0.00283
Read / Write	4	0.0014

C. Request Response Time

Response time for a read or write request is the most critical for measuring the performance of the primary storage system. The current workload that has been experimented with is typically a write-intensive and balanced workload. The deduplication engine will be immensely occupied to serve the requests. However, the read requests from the storage client should be served with minimal latency. Reducing the metadata overhead by leveraging the block similarity has helped improve the performance of both read and write request. Table IV and Table V describe the response time of write and read requests, respectively. The time described in the below tables includes only the access time of metadata and construction or deduplication of a file. It also includes the time to read the metadata from the disk during a cache miss. It does not include the time involved in calculating the fingerprint for the content and the write-back time from the cache to disk. The workload-dependent cache eviction model built on top of the deduplication system yields higher hit rate resulting in substantial time-saving response time.

TABLE IV
WRITE RESPONSE TIME

Superblock Size (KB)	Average Response Time (ms)
32	15.483
64	38.41
128	57.326
256	96.112

TABLE V
READ RESPONSE TIME

Superblock Size (KB)	Average Response Time (ms)
32	5.23
64	11.324
128	19.553
256	25.612

From the above tables it is evident that response time increases as we increase the superblock's size. As the superblock size increases, the number of blocks within the superblock increases. During deduplication, the number of metadata comparisons increases, increasing response time. Though the data fragmentation issue is minimal with larger superblock sizes, response time and number of writes eliminated are higher in smaller superblock sizes.

D. Metadata Overhead

The objective of the project is to keep the number of metadata lookups to be minimal. Experiments were conducted to understand the number of metadata operations involved for every block read or write in a system enabled with deduplication. Fig. 11. depicts the average number of metadata operations for a sequence of blocks. The results help in understanding that the categorization of superblocks helps in storing the shared metadata. During deduplication, we can narrow the duplicate comparison only with the

metadata belonging to a particular category. The system without categorization involves an extensive search throughout the database for relevant metadata during write or read operation. This increases the overhead of metadata in a system without categorization. However, in the homes block I/O trace, the number of metadata operations is slightly higher since most of the LBA's are continuously modified and updated, leading to the creation of new fingerprints.

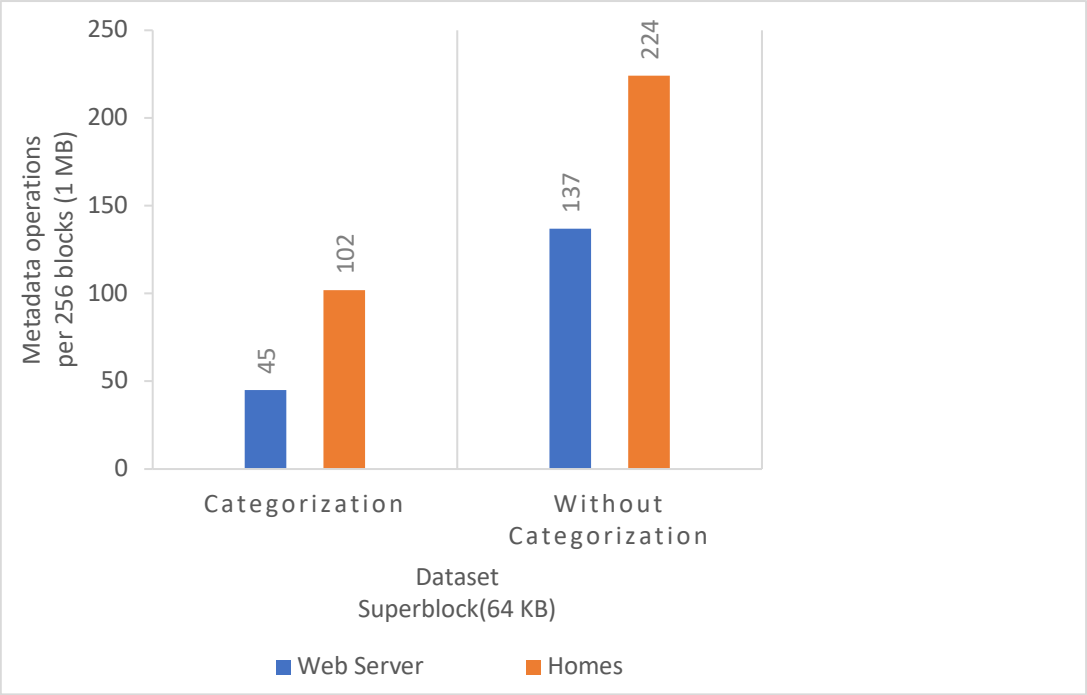

Fig. 11. Metadata overhead

E. Write Elimination

Elimination of duplicate writes will help in improving the performance of the I/O path. Fig. 12. depicts the percentage of duplicate writes that have been eliminated by the deduplication system for two different datasets. However, certain duplicate writes were not eliminated due to the constraint in fragmentation of the data. As mentioned in the earlier sections, a threshold value should be satisfied to perform deduplication on the write request. If the threshold is not met, the request is executed without deduplication. Though

there are few duplicate writes that have not been eliminated, the performance gain from reducing the data fragmentation is huge. As the size of the superblock grows, the amount of duplicate writes elimination decreases.

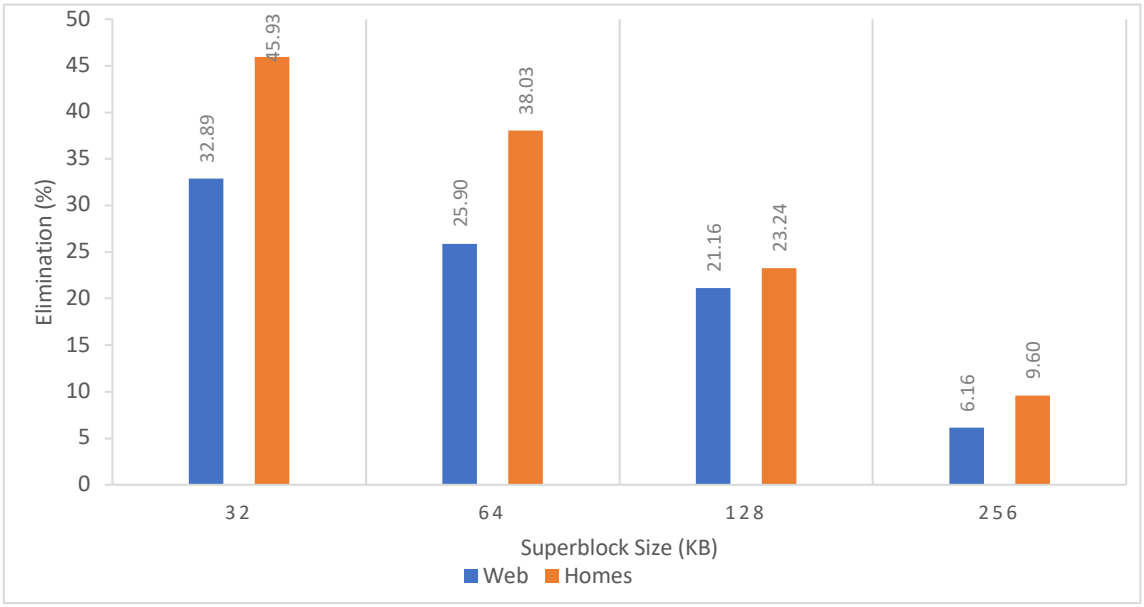

Fig. 12. Eliminated write requests

F. Throughput Analysis

One of the important measures to primary storage performance is throughput. It is important to exhibit higher throughput in an actively accessed storage environment. Due to the absence of a unified test environment, it is relatively difficult to compare the throughput for multiple systems. The throughput depends on the hardware, deduplication ratio, cache hit rate, and percentage of duplicate write eliminated. The exploitation of the similarity between the blocks contributes to higher throughput. The critical component for performance is measuring the caching performance of the system. We have investigated several caching algorithms that are in practice for the primary storage system. The workload that has been tested is write intensive and balanced. The cache is built with a Write-back

strategy. The updates to the elements present in the cache are updated only in the cache and are not written onto the disk till the element is flushed out of the cache. This helps in reducing the disk access for the elements in the cache. The time taken to flush the data marked with the dirty bit is not considered in the initial analysis. The time taken to write the data and metadata updates into the disk is not considered during the below throughput analysis. It measures the efficiency of different caching schemes.

The below figures Fig. 13. and Fig. 14. depicts the analysis of throughput between different systems enabled with different cache eviction policies. From the figures, the throughput is higher with smaller-sized superblocks since the overhead in the deduplication process is lower as compared with larger block sizes and the number of smaller requests that are processed are lower with smaller superblock size. An increase in the number of smaller requests will increase the number of metadata lookups thereby increasing the processing time.

A system design with the concept of categorizing similar superblocks supported by ML-based cache eviction policy with a pre-trained KNN model and a Bloom filter yields a higher throughput than LRU and LFU cache eviction strategy. Higher cache hit rates and duplicate elimination percentage contribute to the higher throughput of the system. The below section describes the hit rate analysis between several cache eviction policies. Though the hit rate is significant in deduplication, the processing time of the cache eviction policy plays a vital role in the throughput of the system. The results are shown in fig. 15. and fig. 16. shows that the hit rates of ML-based cache eviction policy are higher than LRU and LFU. However, the processing time of the RF model in choosing the eviction candidate is significantly higher than the KNN and LRU. During the throughput analysis, a system enabled with the RF model has lower throughput than a system with a lower hit rate such as the LRU

eviction policy. Hence, the RF model is not suitable for primary storage deployment. On the other hand, the KNN model processes faster and it has a higher hit rate thereby witnessing a higher throughput than a cache enabled with the LRU and LFU policy.

The throughput of LRU is higher in 128 KB superblock size since the hit rate of ML-based eviction policy is only slightly higher than LRU model. From a performance perspective, LRU performs better than ML-based eviction policy for larger superblock sizes. As the size of the superblock increases, the number of categories, amount of data to train the ML-based model's decreases resulting in a poorer hit rate. We could achieve a maximum cache component throughput of 13.478 MB/S for 21 days of real-world workload. The cache component throughput is at least 14.4% better with ML-based eviction policy as compared with LRU and LFU eviction policy in a write-intensive workload.

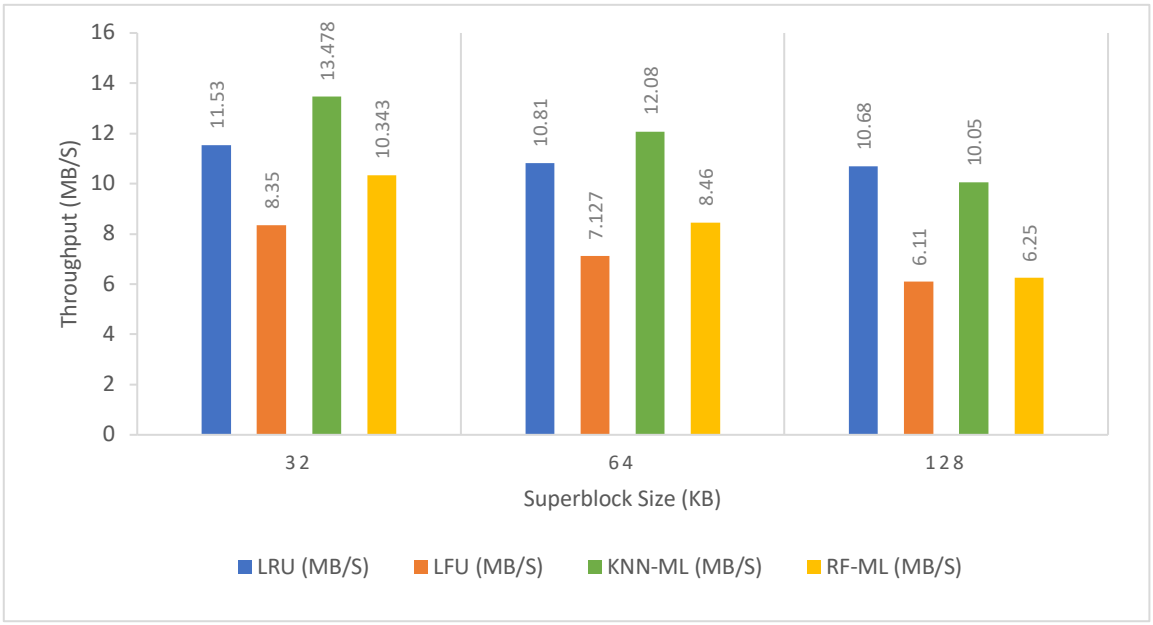

Fig. 13 Caching Policy vs Throughput Analysis

The overall throughput includes the processing time of storing the metadata, file to disk, caching, deduplication. The system with the above-mentioned setup could achieve a maximum overall throughput of 3.74 MB/S for 21 days of real-world workload. The overall

throughput is at least 22.19% better with ML-based eviction policy as compared with LRU and LFU eviction policy in a write-intensive workload. The below analysis supports the fact that the throughput of the system is dependent on the underlying hard disk. The overall throughput drops because of writing the metadata and file into the disk.

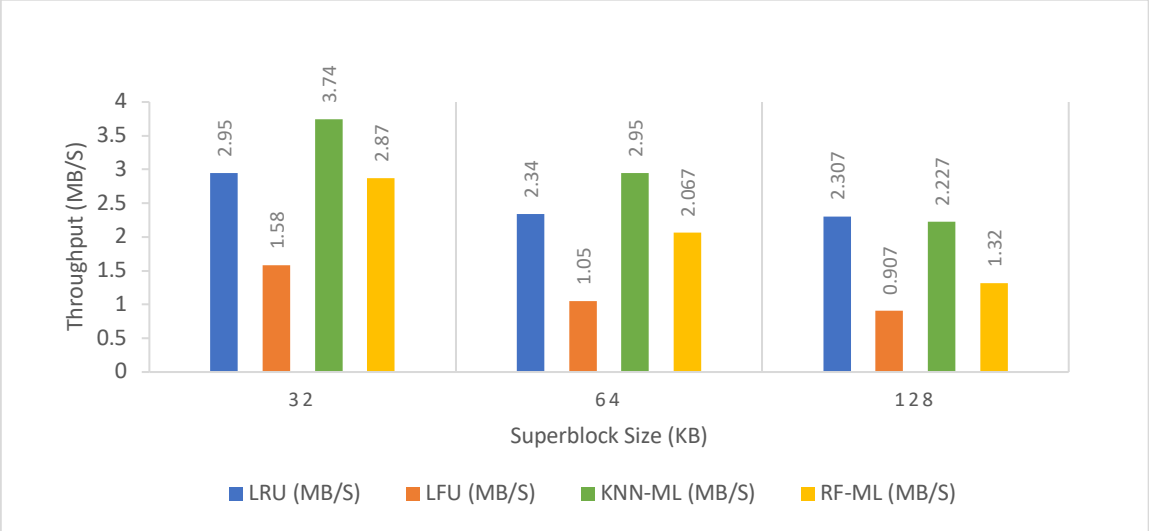

Fig. 14. Overall Throughput Analysis

G. Cache Analysis

Effective cache management will help in improving the performance of the deduplication system. Belady's algorithm is considered as a baseline to understand the performance of the traditional and workload-dependent ML cache eviction strategy. The ML algorithm has experimented with multiple hyperparameters. The ML model will predict the eviction probability of each item in the cache by learning the statistics of past cache misses and considering the current instance of the cache. The experimentation setup includes 15 MB cache, 0.3 * number of items in the cache as eviction count, four features providing various statistics for items in the cache, sampling the data for every number of items in the cache to avoid overfitting the ML model. The experiment was conducted by warming up the

cache with 19 days of block I/O and measuring the hit rate in the cache by various algorithms by sending two days of block I/O. Below are the results for the setup.

1) Belady's algorithm: The optimal hit rate that can be achieved for the above set is 63.046%. This hit is considered as the baseline for measuring the performance of the other algorithms.

2) LRU: The maximum hit rate that the LRU cache eviction policy can achieve is 47.297%. The processing time of LRU strategy for two days of block I/O is 1.14 minutes.

3) LFU: The maximum hit rate that the LFU cache eviction policy can achieve is 42.41%. The processing time of LFU strategy for two days of block I/O is 2.09 minutes.

4) Traditional vs. ML eviction strategy: From the results obtained from traditional and workload-dependent ML based eviction policies, we have analyzed the performance of both methods in terms of hit rate and processing time. Below Fig. 15. describes the efficiency achieved by traditional and ML eviction strategies with respect to hit rate and Fig. 16. describes the processing time. The cache allocated for the metadata information for this experiment is 15MB to mimic the actual cache in a storage system.

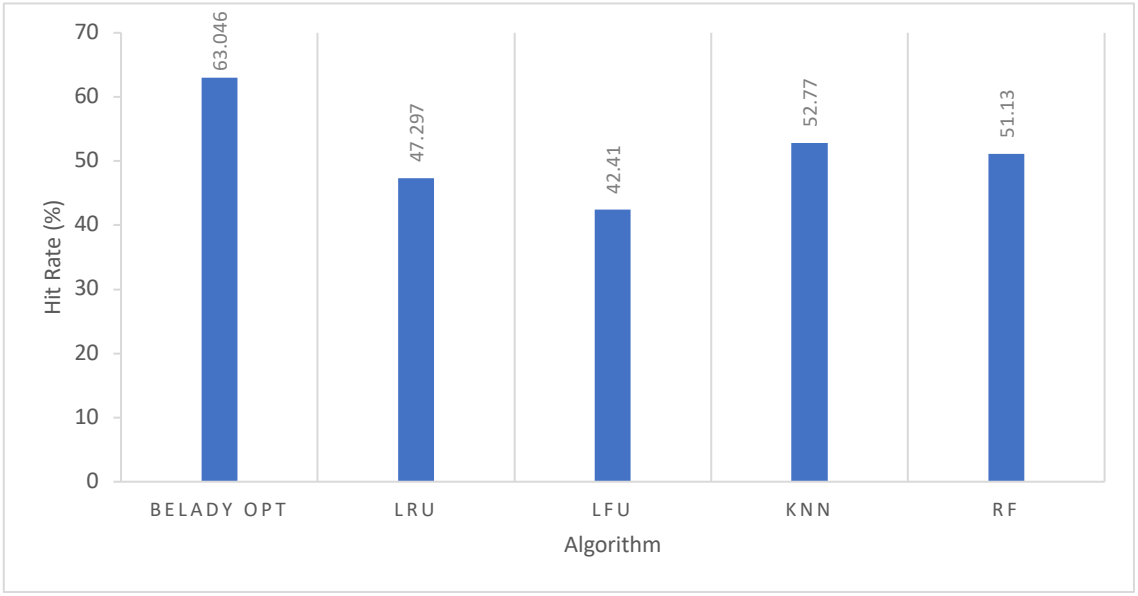

Fig. 15. Traditional vs. ML eviction strategy hit rate analysis

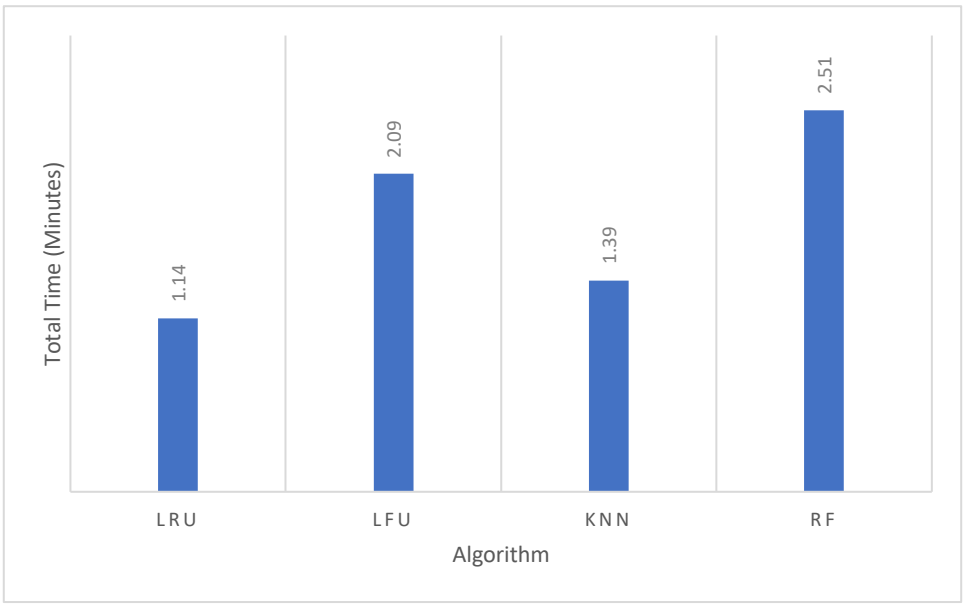

Fig. 16. Traditional vs. ML eviction strategy time analysis

The results obtained from the above experiments show that the cache hit rates based on the workload-dependent Machine Learning model are higher by 5.43%,10.36% over LRU eviction and LFU eviction policy respectively with a metadata cache allocation of 10% of average everyday working data stream size. The cache system learns the past evicted block I/O statistics and refine itself while choosing an eviction candidate, thereby performing better than traditional cache eviction policies. Though the processing time of ML based eviction policy is slightly higher than the traditional approach, it is acceptable to adopt ML based eviction policy because of higher hit rates. The time taken to process block I/O on account of a cache miss is much higher than the processing time of ML based eviction model. ML based cache eviction policies can be deployed to the primary workload, and therefore, it is wise to adopt ML based cache eviction policy for the current system. For the practical reasons, the amount of time taken to process the blocks in a ML-based model should be minimal. From the results shown in fig. 15. and fig. 16., it is evident that a pre-trained KNN model will be the best choice for a primary storage deduplication system. It is

mainly the choice because of the throughput gain as depicted in fig. 13. and the hit rates. Though the RF model has higher hit rates as compare with LRU, it performs slower and has lower throughput than the system with LRU eviction policy. It is mainly because of multiple individual decision trees that are involved in choosing an eviction candidate. The time involved in the RF model is relatively higher and thus there is a degradation in throughput performance.

5) *Cache Size Analysis:* The ML-based cache is workload-dependent, and it is crucial to find the correct size of the cache for storing the metadata information. The size of the cache was decided on the average workload metadata statistics obtained over 19 days. An analysis of the cache size for metadata and hit rates was made. The results are shown in fig. 17. From the below figure, it is evident that as we increase the size of the metadata cache, the hit rate increases. However, this cache refers to only the metadata cache and not the actual data cache. The hit rate increases as we increase the size of the metadata cache. However, there is a significant increase in hit rate when the cache size is increased from 5% to 10% of average metadata. The hit rate does not seem to be higher as we double the metadata cache size from 10% to 20%. The candidate cache size for the real-world workload that was experimented with is 10% of everyday metadata size. The cache hit rates based on the workload-dependent Machine Learning model are higher by 5.43%,10.36% over LRU eviction and LFU eviction policy respectively with a metadata cache allocation of 10% of the average everyday working data stream size.

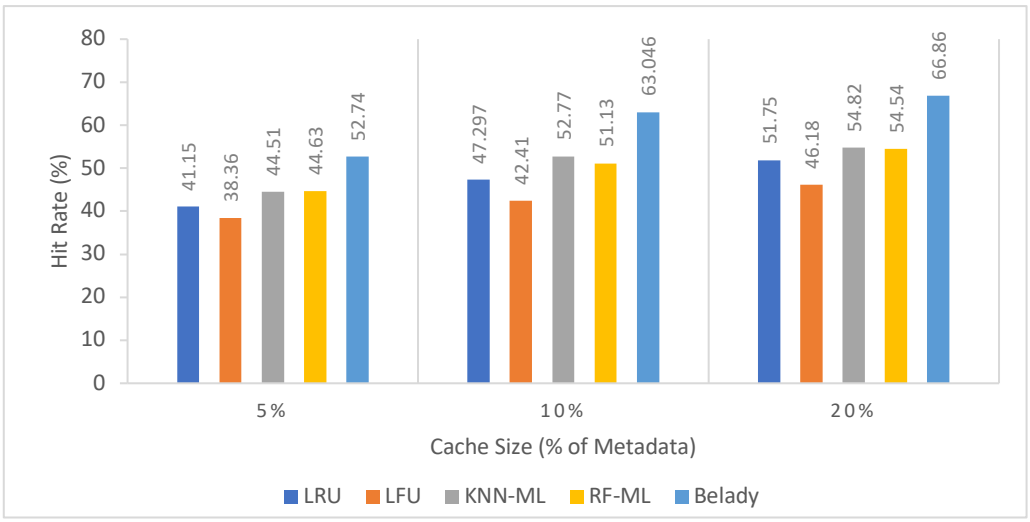

Fig. 17. Cache Size vs Hit Rate Analysis

6) KNNC: The number of neighbors is a hyperparameter that can be varied to yield better results. The performance of the ML model is measured based on the Hit rate achieved by the model and the time it takes to process the block I/O of two consecutive days. Fig. 18. describes the hit rate for various hyperparameter setups. The highest hit rate we can achieve through a KNNC for web server dataset is 52.77%. Table VI describes the time involved in training and testing of the KNNC model. As we increase the number of neighbors, the training and testing time tends to drop. When the number of neighbors is reduced, the complexity of the KNN technique increases since it must run more regularization or smoothing. Assuming if the number of neighbors is 1, each data point becomes the center of an eviction candidate class and noneviction candidate class. The points will become intertwined. Thus, it becomes difficult to differentiate between the classes for a point which also increases the complexity. However, when we increase the number of neighbors, the classification area becomes more smoother, and the class of the point is decided by the majority from the nearest neighbors. As the area of nearest neighbor increases, it becomes less complex to classify an eviction and noneviction candidate class.

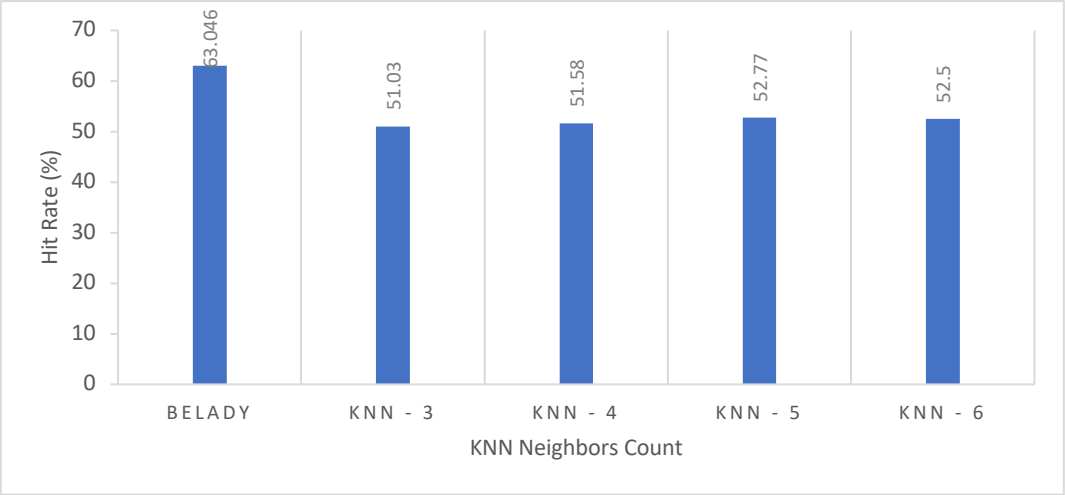

Fig. 18. KNNC hit rate analysis

TABLE VI
KNNC TIME ANALYSIS

Neighbors	Training Time (min)	Testing Time (min)
KNN - 3	28.33	2.04
KNN – 4	26.14	1.55
KNN – 5	22.07	1.39
KNN – 6	21.59	1.18

7) RFC: The no_of_estimators, max_depth, min_sample is a hyperparameter that can be varied to yield better results. Fig. 19. describes the hit rate for various hyperparameter setups. The highest hit rate we can achieve through RFC for web server dataset is 51.13%. Table VII describes the time involved in training and testing of the RFC model. Though multiple hyperparameter combinations were tested, we have considered only the series of experiments that yield the best results. The RFC setup in the below figure can be read as RF (no_of_estimators, max_depth, min_sample). In RFC, the model can be improved by increasing the number of decision trees, however, the cost of the model grows as we increase the number of decision trees. Another hyperparameter is the depth of the tree. More information is conveyed to the decision trees if we increase the depth of the

trees. A more precise information about the dataset is conveyed to the decision trees for classification between an eviction and noneviction candidate class with larger depth. From the below Fig. 19. we can see that the hit rates increases as we increase the depth of the decision trees. However, the computational cost increases as we increase the depth of the tree. Table VII shows that the training and testing time of a RFC model increases with an increase in the depth of the tree.

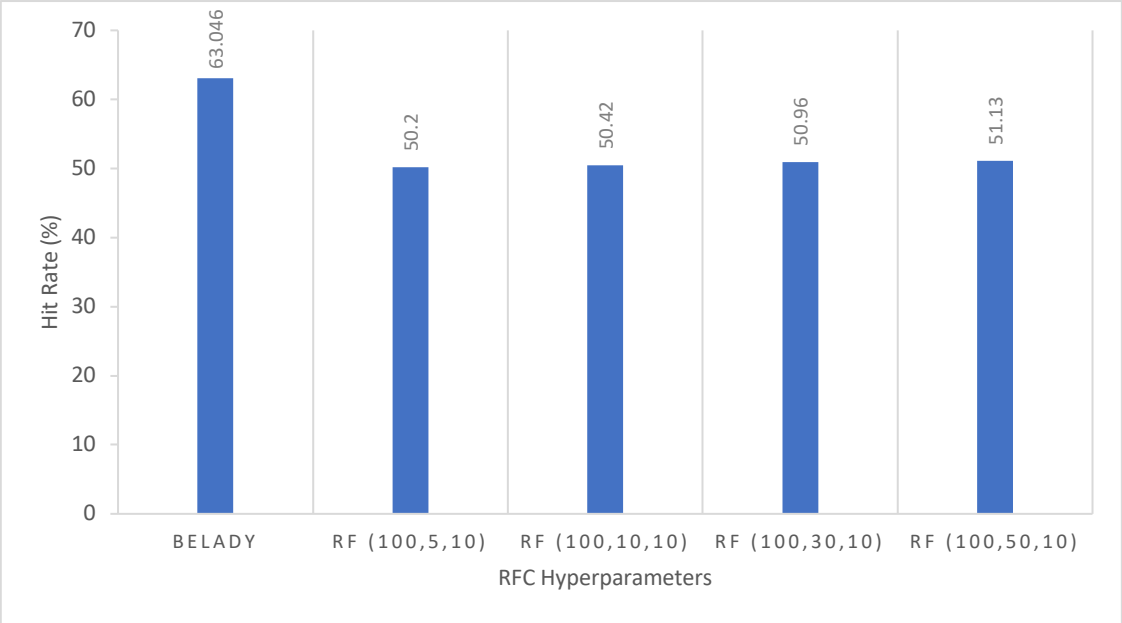

Fig. 19. RFC hit rate analysis

TABLE VII
RFC TIME ANALYSIS

Hyperparameters	Training Time (min)	Testing Time (min)
RF (100,5,10)	40.56	2.22
RF (100,10,10)	41.18	2.41
RF (100,30,10)	45.27	2.45
RF (100,50,10)	50.16	2.51

8) Eviction count: Eviction count provides us the number of eviction candidates that can be generated for eviction when the cache is full. It is one of the hyperparameters for the ML based eviction setup. Based on the size of the eviction count, there is a change in the hit

rates for the ML based model. Fig. 20. depicts the relationship between the eviction count and hit rate. The trend shows that the hit rate gets decreased when the number of eviction count increases. However, for the current setup, an eviction count of 0.3 * number of elements in cache yields the best result.

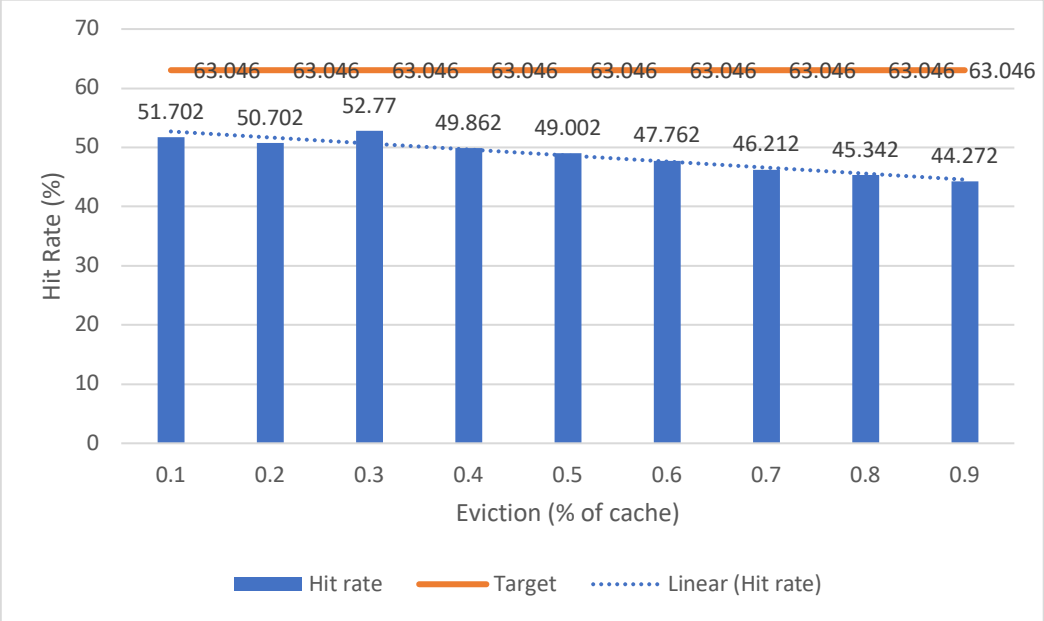

Fig. 20. Eviction count vs. hit rate analysis

9) Complexity Analysis: The above experiments show that implementing a ML-based cache eviction policy results in increased hit rates and overall throughput compared to a system with LRU and LFU cache eviction policy. There exists complexity in building a system with a workload-dependent Machine Learning model to understand the workload statistics and choose the right eviction candidate. The ML model runs as a background process with continuous learning from the incoming block I/O requests. The model refines and gets better as the number of input data increases. The time involved in processing the eviction candidate by the model is another important factor contributing to the system's overall throughput. The implementation of the ML model is resource-intensive, depending on the model hyperparameters and workload. The time involved in processing

two days of block I/O requests with different caching algorithms is shown in Fig. 21. The figure shows that the LRU cache eviction system processes the blocks faster than any other eviction policy in our setup. Though the processing time of KNNC is slightly higher, the benefits are realized with higher hit rates of KNNC compared with LRU eviction system as shown in Fig. 13., Fig. 14. and Fig. 17. Also, as we increase the metadata cache size, the processing time increases due to an increase in the number of elements in the cache.

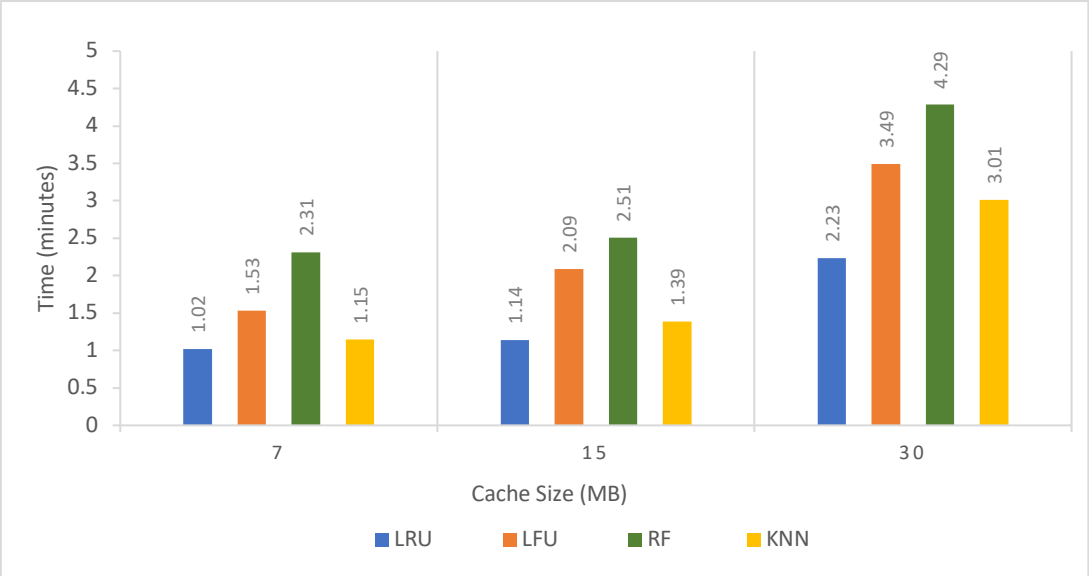

Fig. 21. Cache size vs. block processing time

From the above tables VI and VII, it is evident that the time taken to train and yield eviction candidates for RFC model is higher than the KNNC model. Therefore, the overall throughput is affected based on the model and the hyperparameters of the model. The hit rate of KNNC is slightly higher than RFC for this setup; We can choose KNNC model for having better performance. Though the background learning process of the model utilizes additional computation, we can realize benefits with higher hit rates and lower eviction candidate processing time. Therefore, implementing ML-based cache eviction depends on the choice of the model, resource utilization, hit rate and processing time of the model compared with traditional cache eviction techniques.

H. Superblocks and Categorization

In the experiments conducted with two production systems, the I/O traces are passed to the deduplication engine, where the incoming data stream is split into superblocks and categorized based on the similarity. Table VIII shows the relation between size of the superblock and the number of categories.

TABLE VIII
SUPERBLOCK VS. CATEGORY

	Web Server Block I/O			Homes Block I/O		
Size (KB)	**# of categories**	**Duplicate**	**Duplicate %**	**# of categories**	**Duplicate**	**Duplicate %**
32	608194	254458	29.5	958924	549852	36.43
64	237997	78083	24.7	396980	196936	33.12
128	97199	25379	20.7	121288	30062	19.86
256	14149	1554	9.01	25056	3513	12.6

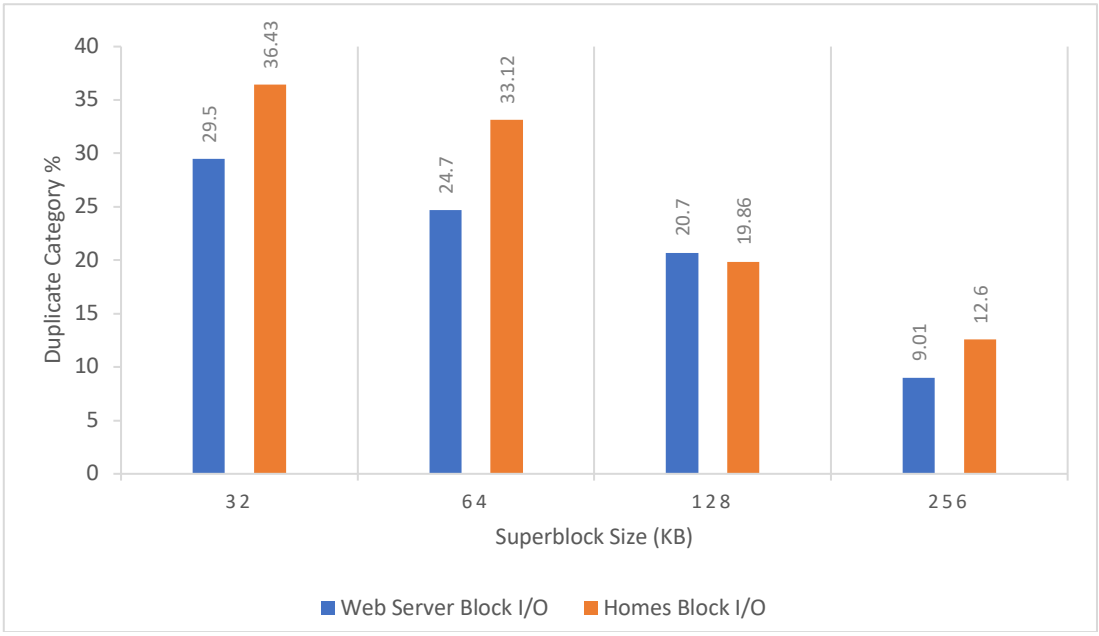

Fig. 22. Duplicate percentage vs. superblock size

From the above table, we can see that the categorizing the incoming blocks helps categorize similar blocks into a single category. Fig. 22. describes the presence of a certain

percentage of superblocks having duplicate blocks. The duplicate percentage increases as the superblock size decrease.

I. Bloom Filter Significance

Bloom filters serve as an important component during the categorization. As mentioned in the earlier section, while creating a new category, SBR is passed on to the Bloom filter and check if there is a category already present. We evaluated the Bloom filter data structure with a hash table for looking up if a category is already present. We know that hash table structure yields a quicker response during a search operation; the performance of the data structure degrades as the data size grows. This key observation has been captured in Fig. 23. The figure depicts the analysis obtained from the Bloom filter implementation to web server block I/O tracer dataset.

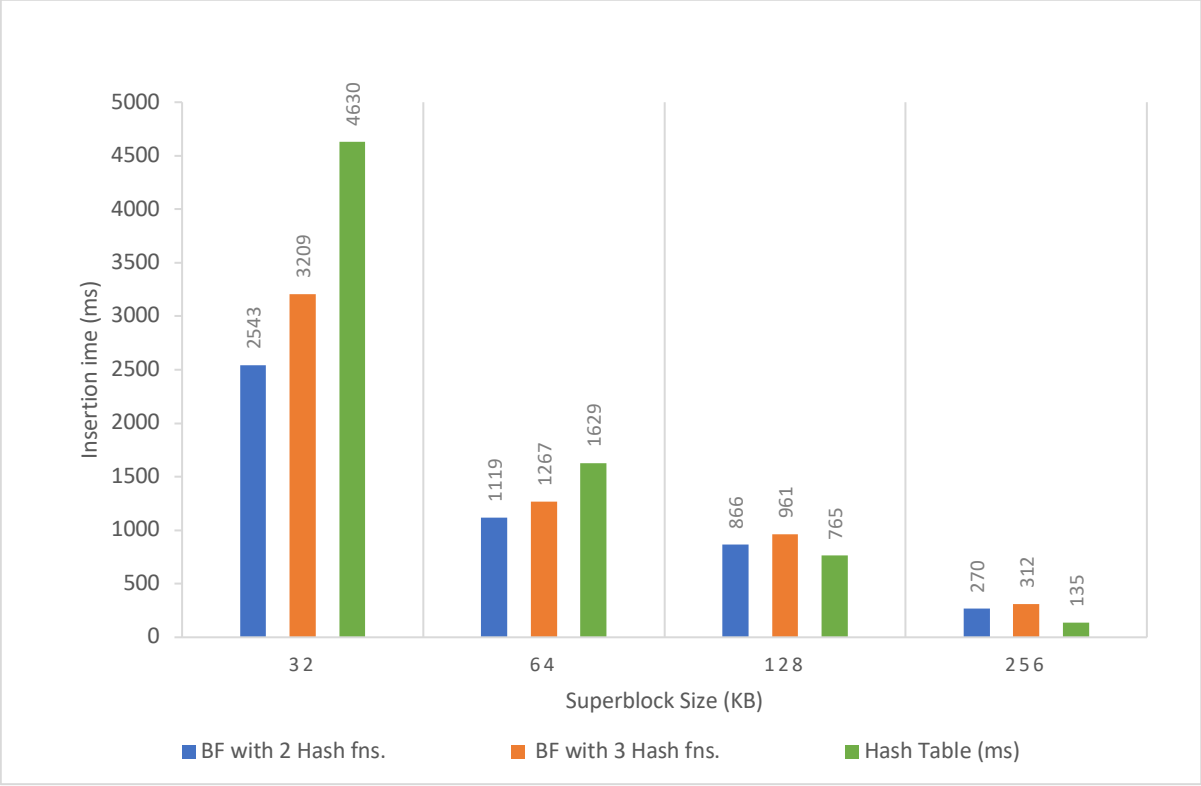

Fig. 23. Category insertion time analysis

The above figure helps in understanding the trend of the insertion time when we use different data structures. The insertion time of the Bloom filter performs much better than the hash table as data grows. Since insertion time is the most time-consuming activity in the BF, it is advisable to activate BF for the deduplication system when the number of categories increases beyond 100K entries. Fig. 24. helps in understanding the space saving trend between different data structures. Bloom filters do not store the data. Therefore, the Bloom filter size can incur constant space based on the False Positive Rate (FPR) and number of entries. We have used a Bloom filter of same size (1.5 MB) for all the superblock sizes. However, the trend is nearly exponential when we use the hash tables since the actual data is stored in the tables. From a memory perspective, it is beneficial to use a Bloom filter in deduplication engine for primary storage workload.

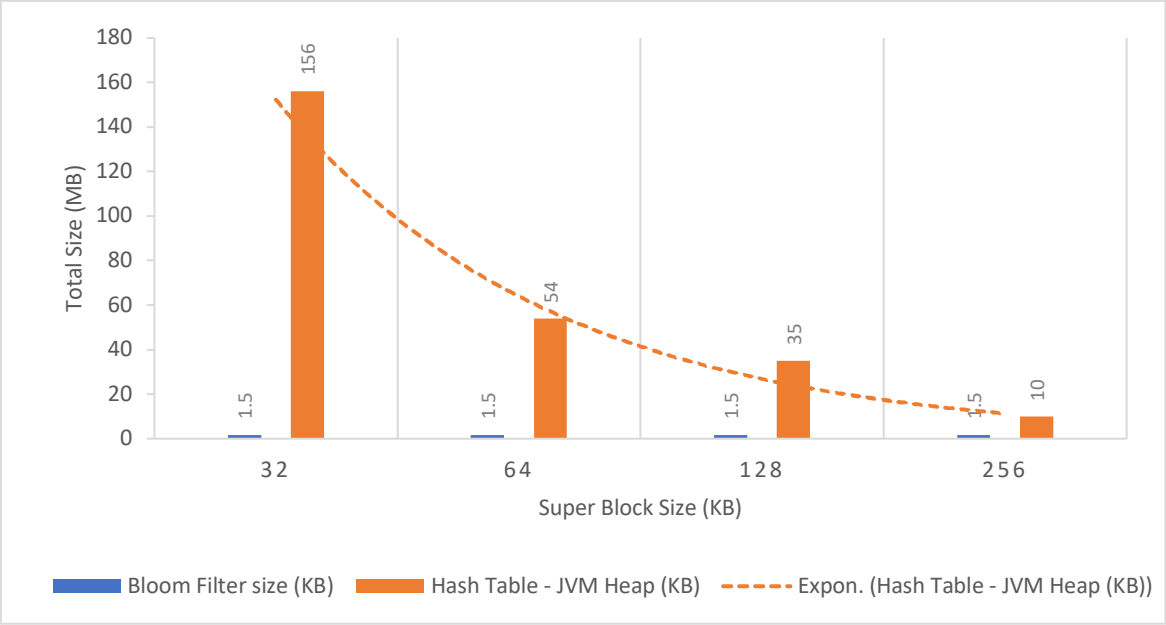

Fig. 24. Category space analysis

From the results, it is evident that the usage of BF is significantly practical when the size of the superblock is 64 KB or below. In our setup, we configured BF for the superblock size of 64 KB. BF is a probabilistic data structure, and it might incur false positives. The FPR is

less than 1%, and it can be ignored for the configuration mentioned above. We can maintain a lower FPR if the size of BF is increased.

VI. CONCLUSION AND FUTURE WORK

The above system leverages the similarity between the blocks to build a deduplication system with lower metadata overhead and higher throughput for primary storage. The incoming data block request is split into multiple superblocks and categorized based on the Broder's theorem. The categorization narrows the metadata search during deduplication resulting in lower metadata I/O's and eliminating redundant write requests. A real-world block I/O trace is used for the evaluation of the system. The result from the experiments shows that the elimination of duplicate writes by the system can be as high as 32.89% and 45.93% in web server and homes block I/O traces. The average response time of both the read and write requests of the newly designed deduplication system is around 5.23 ms and 15.483 ms respectively. The overall throughput gain is at least 14.4% better with ML-based eviction policy as compared with LRU and LFU eviction policy in a write-intensive workload. The number of metadata I/O's has reduced significantly by building an efficient ML based cache eviction strategy, leveraging superblock similarity and categorization. Hence the newly built system can be implemented for a primary storage system.

Following entities achieve an efficient metadata management system for deduplication:

1) An efficient workload-dependent ML based cache eviction strategy is designed for the write-intensive and balanced workload with varying parameters like eviction count and sampling frequency, the ML based cache eviction strategy have hit rate higher by 5.43%,10.36% over LRU eviction and LFU eviction policy respectively with a metadata cache allocation of 10% of average everyday working data stream size.

2) Similarity detection algorithm was built to identify similar superblocks and share

their metadata information, reducing the number of lookups while deduplicating the incoming block of the same category.

3) Reducing the disk fragmentation issue by deduplication only the superblocks that satisfy a threshold value for the match percentage.

4) We have implemented a Bloom filter in the system for reducing the number of disk I/O in new category creation or identification.

The future directions for this system are:

1) Currently, this system works only on a single node environment. The future aim is to build a deduplication system for a multi-node environment.

2) Improving the workload-dependent ML based cache eviction strategy by hyperparameter tuning and prefetching blocks based on category access pattern to reach hit rate nearer to optimal cache hit rate.

3) Studying and understanding the locality of the blocks after deduplication and reorganizing to preserve block locality.

REFERENCES

[1] T.Coughlin, 175 zettabytes by 2025, Nov. 2018. Accessed on: Aug. 06, 2020. [Online]. Available: https://www.forbes.com/sites/tomcoughlin/2018/11/27/175-zettabytes-by-2025/?sh=265c0c315459.

[2] Z. J. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, “Cluster and Single-Node Analysis of Long-Term Deduplication Patterns,” *ACM Transactions on Storage*, vol. 14, no. 2, pp. 1–27, 2018.

[3] A. El-Shimi, R. Kalach, A. Kumar, J. Li, A. Oltean and S. Sengupta, "Primary Data Deduplication—Large Scale Study and System Design", *Proceedings of the 2012 conference on USENIX Annual Technical Conference*, pp. 285-296, 2012.

[4] D. T. Meyer and W. J. Bolosky, “A study of practical deduplication,” *ACM Transactions on Storage*, vol. 7, no. 4, pp. 1–20, 2012.

[5] EMC, "Achieving storage efficiency through EMC Celerra data deduplication". EMC white paper, March 2010. [Online]. Available: https://www.dell.com/community/s/vjauj58549/attachments/vjauj58549/celerra/660/1/h6065-achieve-storage-effficiency-celerra-dedup-wp.pdf.

[6] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression", in *IEEE Transactions on Information Theory,* vol. 23, no. 3, pp. 337-343, May 1977.

[7] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding", in *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530-536, Sept. 1978.

[8] H. Yu, X. Zhang, W. Huang, and W. Zheng, “PDFS: Partially Dedupped File System for Primary Workloads,” in *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 863–876, Mar. 2017.

[9] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, “Extreme Binning: Scalable, parallel deduplication for chunk-based file backup,” *in IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 1–9, 2009.

[10] A. Godavari, C. Sudhakar and T. Ramesh, "Hybrid Deduplication System — A Block-Level Similarity - Based Approach," *in IEEE Systems Journal*, pp. 1-11, 2020.

[11] X. Du, W. Hu, Q. Wang, and F. Wang, “ProSy: A similarity based inline deduplication system for primary storage*,”* in *IEEE International Conference on Networking, Architecture and Storage (NAS), pp*. 195–204, 2015.

[12] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage*",* in *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST)*, pp. 89-101, 2002.

[13] L.A. Belady, "A study of replacement algorithms for virtual-storage computers*", IBM Systems Journal*, vol. 5, no. 2, pp. 78-101, 1966.

[14] Tony Yiu, Understanding Random Forest, Jun. 2019. Accessed on: June 07, 2020. [Online]. Available:https://towardsdatascience.com/understanding-random-forest-58381e0602d2

[15] “Scikit-learn: machine learning in Python,” Jun. 2008. Accessed on: Dec. 02, 2020. [Online]. Available: https://scikit-learn.org/stable/

[16] FIU traces web-link. 2010. Accessed on: June 07, 2020. [Online].Available: http://iotta.snia.org/traces/390/

[17] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, “idedup:latency-aware, inline data deduplication for primary storage,” in *Proceedings of the Tenth USENIX Conference on File and Storage Technologies (FAST)*, vol. 12, pp. 1–14, 2012.

[18] W. Dong, F. Douglis, K. Li, H. Patterson, S. Reddy, P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters", in *Proceedings of the Ninth USENIX Conference on File and Storage Technologies (FAST)*, pp. 15–29, 2011.

[19] Y. Fu, H. Jiang, and N. Xiao, “A scalable inline cluster deduplication framework for big data protection”, in *Proceedings of the 13th International Middleware Conference,* pp. 354–373, 2012.

[20] G. Grangia, Q. Xu, A. Bianco and P. Giaccone, "Balancing the storage in a deduplication cluster", in *IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1-4, 2017.

[21] A. Khan, C. Lee, P. Hamandawana, S. Park and K. Youngjae, "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems", in *IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 87-93, 2018.

[22] C. Alvarez. "NetApp deduplication for FAS and V-Series deployment and implementation guide". Technical Report TR-3505, NetApp, 2011.

[23] B. Mao, H. Jiang, S. Wu, and L. Tian, “Leveraging data deduplication to improve the performance of primary storage systems in the cloud,” in *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1775–1788, Jun. 2016.

[24] A. Wildani, E. L. Miller, and O. Rodeh, “Hands: A heuristically arranged non-backup in-line deduplication system,” in *IEEE 29th International Conference on Data Engineering (ICDE)*, pp. 446–457, 2013.

[25] “BloomFilter (Guava: Google Core Libraries for Java 23.0 API,” Nov. 2020. Accessed on: Dec. 17, 2020. [Online]. Available: https://guava.dev/releases/23.0/api/docs/com/google/common/hash/BloomFilter.html

[26] “Home | Jython,” Feb. 2001. Accessed on: Feb. 13, 2021. [Online]. Available: https://www.jython.org/

[27] H. Wu, C. Wang, Y. Fu, S. Sakr, K. Lu and L. Zhu, “A differentiated caching mechanism to enable primary storage deduplication in clouds,” in *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 6, pp. 1202–1216, Jun. 2018.

[28] A. Broder, "On the resemblance and containment of documents", in *IEEE Proceedings Compression and Complexity of Sequences Conference (SEQUENCES '97)*, pp. 21-29, 1998.

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