Design of Primary Storage Deduplication Schemes for Cloud Environment

Submitted in partial fulfillment of the requirements for

the degree of

DOCTOR OF PHILOSOPHY

Submitted by

Amdewar Godavari Ramlu

(Roll No. 716175)

Under the guidance of

Dr. Chapram Sudhakar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL
TELANGANA - 506004, INDIA

April 2022

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF TECHNOLOGY WARANGAL
TELANGANA - 506004, INDIA

CERTIFICATE

This is to certify that the thesis titled, Design of Primary Storage Deduplication Schemes for Cloud Environment, submitted in partial fulfillment of requirement for the award of degree of DOCTOR OF PHILOSOPHY to National Institute of Technology Warangal, is a bonafide research work done by Mrs. Amdewar Godavari Ramlu [Roll No. 716175] under my supervision. The contents of the thesis have not been submitted elsewhere for the award of any degree.

Research Supervisor
Dr. Chapram Sudhakar
Associate Professor
Dept. of Computer Science and Engg.
NIT Warangal
India

Place: Warangal
Date:

DECLARATION

This is to certify that the work presented in the thesis entitled "*Design of Primary Storage Deduplication Schemes for Cloud Environment*" is a bonafide work done by me under the supervision of Dr. Chapram Sudhakar. The work was not submitted elsewhere for the award of any degree.

I declare that this written submission represents my ideas in my own words and where others ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Amdewar Godavari Ramlu
(Roll No. 716175)

Date:

ACKNOWLEDGEMENTS

Every day during my Ph.D. has been a great opportunity for learning. This thesis work is the result whereby I have been supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

First and foremost, I want to thank God for blessing me with all the strength and health I need to complete my research. It is with great pleasure that I acknowledge my sincere thanks and deep sense of gratitude to my supervisor Dr. Chapram Sudhakar, Associate Professor, Department of Computer Science and Engineering, National Institute of Technology (NIT) - Warangal for his valuable guidance throughout the course. His technical perception, profound knowledge, sincere effort in guiding a student and devotion to work have greatly charmed and constantly motivated me to work towards the goal. He always gave me ample time for discussions, reviewing my work and suggesting requisite corrections.

I extend my gratitude to the Doctoral Scrutiny Committee (DSC) members comprising of Prof. S. G. Sanjeevi, Dr. S. Ravichandra, Dr. Rashmi Ranjan Rout and Prof. D. Shrinivascharya for their insightful comments and suggestions during oral presentations. I am also grateful to Prof. T. Ramesh who has always been supportive and encouraging all through my tenure. I am lucky to attend lectures by Prof. B. B. Amberker and Dr. Ch. Sudhakar during my tenure. I am immensely thankful to Dr. Ch. Sudhakar, Prof. R. B. V. Subramanyam, Prof. P. Radha Krishna and S. Ravichandra Heads of Dept. of CSE and chairmans of DSC, during my tenure for providing adequate facilities in the department to carry out the oral presentations.

I wish to express my sincere thanks to Prof. N.V. Ramana Rao, Director, NIT Warangal for providing the infrastructure and facilities to carry out the research. I am also very much grateful to the faculty members of Computer Science and Engineering Department for their moral support throughout my research work.

On the personal level, I would also like to thank my scholar friends NIT-Warangal for their valuable suggestions and for extending selfless cooperation. Lastly, my gratitude to my family for their unconditional love, support and prayers for my success in achieving the goal.

Amdewar Godavari Ramlu

Dedicated to

My Family

ABSTRACT

Current technological trends such as Big data, Cloud Computing and Internet Of Things are the main sources of explosive data generation and are using cloud storage for data backup purposes. There is a need for data storage techniques that eliminate duplicates efficiently. Deduplication systems being developed in the recent past are serving as effective solutions to the problem. Deduplication systems adopt chunking of data in fixed size or variable size data blocks. These data blocks undergo fingerprint computation using SHA-1 or MD-5 hashing algorithms, index lookup for the computed fingerprint is performed to identify duplicate data blocks and based on this, a decision is taken to store the data block or not.

The thesis focuses on primary storage deduplication schemes for the cloud environment. Primary storage systems have low data redundancy, strict latency constraint and random access patterns. Applying deduplication to the primary storage system results in disk-bottleneck and data fragmentation problems. Deduplication metadata has poor data locality and hashes have random values, and causes frequent disk accesses for metadata. This is known as the disk-bottleneck problem. If each duplicate data is eliminated from incoming write request, sequential data is scattered, known as data fragmentation. In this thesis, the proposed approaches have achieved better response time and storage optimization by maintaining similarity-based indexing and performing selective deduplication.

Firstly, a centralized primary storage deduplication system implemented at the block level is proposed. In this work, requests are categorized as small read, large read, small write and large write requests. Deduplication metadata is maintained separately for small and large requests. Selective deduplication is applied to large write requests. If the deduplication results in data fragmentation beyond a threshold value then duplicate data is not eliminated. Conditional elimination of duplicate data is called selective deduplication.

Secondly, a centralized deduplication system with a content-based cache is proposed. In the cloud, multiple workloads are consolidated to a single machine. Due to interference of workloads, cached locality gets affected. Apart from this, deduplication causes sharing of data blocks, whose frequency and recency cannot be predicted. Traditional cache re-

placement policies are not well suited in this context. Hence, a content-based cache with a Modified-ARC algorithm is proposed.

Thirdly, a centralized primary storage deduplication implemented at file level is proposed. Requests are categorized based on file size and file type. Small size files of all file types, undergo chunk level deduplication. Based on data redundancy in the file, large size files are classified as high duplicate, low duplicate, or unpredictabe duplicate files. Segment level deduplication is used to deduplicate high and unpredictable duplicate files, whereas whole file deduplication is used to deduplicate low duplicate files.

The fourth work is a file semantics aware distributed deduplication system for the cloud. Files are categorized based on data redundancy as high duplicate, low duplicate and unpredictable duplicate files. File type specific chunking and deduplication is performed on files. Deduplication preprocessing (chunking and hashing) is performed at the client. Indexing and lookup are performed at the data server. Highly redundant files and unpredictable files are allocated to the data server using a stateful routing approach and the best server is determined probabilistically. Low duplicate files are assigned to the data server using a stateless routing approach.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The arrival of technologies such as Cloud Computing, Social Networks - Facebook, Netflix, Instagram, Internet of Things, has given rise to data sharing platform. This leads to the need of cloud storage service to store massive amount of digital data and its maintenance. International Data corporation [1] has conducted a survey and they found that the data size will grow beyond 144 ZB by year 2025. Out of this data, 75% of the data is duplicate data. Thus, the main issue to be addressed with cloud storage is to reduce the volume of data to be transferred over network and to be stored on storage devices. IT companies such as Amazon, Google, Microsoft, IBM, Oracle etc. have encouraged research to address the issues related to duplicate data storage. In order to improve storage efficiency, there are two techniques available namely data compression and data deduplication. Data compression technique performs byte by byte or string of bytes comparison in the content of one or few files for identifying duplicate data. Whereas, deduplication technique computes hash value for the content of large size blocks ($\geq$ 512 bytes) and uses it for duplicate data identification among large number of files, which is a better scalable approach compared to compression. Deduplication is the most appropriate technique for large storage systems used in industry as well as academia. Deduplication system is responsible for eliminating duplicate data and storing unique data by creating references to retrieve original data as shown in Figure 1.1. yellow colored block indicates unique data blocks and red colored data blocks indicates duplicate blocks. Thick arrows points towards unique blocks and thin arrows points towards duplicate blocks. While storing data, only unique data blocks are stored and whenever

duplicate data block is seen, reference is added to existing block with same content. This makes data deduplication the most popular with respect to saving on costs.

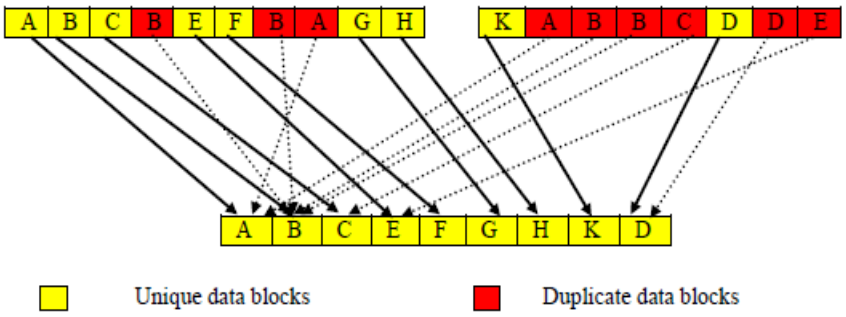


Figure 1.1: Deduplication system example

Deduplication process consists of five phases namely chunking, hashing, indexing, searching, duplicate elimination and storage management. Chunking partitions the given file data into fixed-size or variable-size chunks. To identify the content similarity of chunks, hash value or fingerprint for the content of the chunk is computed using cryptographic hashing algorithms such as Secure Hash Algorithm (SHA-1) or Message Digest (MD-5). Chunk fingerprints of already stored data chunks are stored in index table. In order to determine whether the given data is duplicate, computed fingerprint of the data is searched in the index table. If fingerprint exists, that data is treated as duplicate data. Otherwise, it is treated as unique data and the fingerprint is inserted into the index table. Deduplication eliminates duplicate data blocks and stores unique data blocks which may be referenced by two or more logical blocks. Hence, logical block to physical block mapping and reference count management is required.

The contributions of this thesis are as follows:

- **Hybrid Deduplication System - block level similarity based approach**: Hybrid Deduplication System (HDS) is designed for primary workloads that exhibit random access patterns and weak temporal locality. It works at block level supporting inline as well as offline deduplication. HDS leverages similarity based partial look up to enhance deduplication performance.

- **Hybrid Deduplication System with content-based cache for cloud**: In this work, HDS with content-based data cache is proposed to cope up with random access patterns and weak temporal locality. In order to select the victim for cache replacement, popularity of the content, which is a combination of recency and frequency of references is considered. Deduplication identifies and eliminates duplicate data blocks and enforces data cache to maintain unique blocks.

- **File semantic aware primary storage deduplication system**: In this work, the File Aware DeDuplication system (FADD) is proposed. Files are categorized based on size and file type. Large files can be categorized broadly as high duplicate (H) type, low duplicate (L) type and unpredictable duplicate (U) type. Small files undergo chunk level deduplication. H and U type large files undergo segment level deduplication and L type files undergo file-level deduplication.

- **File aware distributed deduplication system in cloud environment**: In this work, Distributed Deduplication System (DDS) is proposed to perform file type aware, hybrid i.e., stateful and stateless distributed deduplication. Files are categorized based on duplicate data as high duplicate, low duplicate and unpredictable duplicate. High duplicate and unpredictable files undergo superchunk level deduplication. These superchunks are assigned to data servers using stateful routing approach. The best data server for superchunk is selected probabilistically. Whereas, low duplicate files are whole file chunked and assigned to data servers using stateless routing approach.

1.1 Motivation, aim and objectives of proposed work

Recently users of social networks are tremendously growing. These technological trends are being used for data sharing which causes generation of duplicate data. In addition to this, the enterprises also intend to store backup data for handling storage system failures and such backup data contains inherently duplicate content. Their demand for efficient data storage and better I/O performance, have raised the need of cloud storage equipped with deduplication system. Cloud storage systems can be either backup storage or primary

storage. In the past, more research works can be observed in the context of backup storage due to the existence of more duplicate data. However, in recent years, capacity of primary storage systems has been expanded significantly. Primary storage systems store active data which is frequently accessed, and demand for high performance with low latency. The research towards applying deduplication on primary storage systems, has gained abundant attention due to the demand for efficient data storage and improved I/O performance. But primary storage deduplication poses challenges which includes random access patterns, strict latency constraints [2][3], less duplicate data [4][5][6] and contention for resources like CPU and RAM.

Primary storage deduplication systems are designed as either centralized or distributed deduplication systems. In both of these systems, deduplication can be implemented at file level or block level. Block level primary storage deduplication has content only and no other information. Hence, block based deduplication systems can be easily integrated without modifying the existing file systems. However, if low redundancy data such as video files, compressed files etc. are deduplicated at block level, oblivious of their data type, then negligible duplicate elimination with increased deduplication overhead can be observed. However, if data redundancy level of a file is identified, this will help in utilizing the resources efficiently and achieve significant amount of duplicate elimination. At both levels, deduplication can be applied directly on the I/O path of the primary storage systems. Irrespective of the level, applying deduplication raises problems such as extra latency on the I/O path, disk bottleneck and data fragmentation.

The performance of centralized deduplication system can be enhanced by indexing optimization, caching optimizations, selective deduplication etc. Indexing optimizations are locality, similarity or a combination of both similarity and locality based approaches. Different solutions are proposed in the literature, for indexing such as similarity based indexing [3][7], locality based fingerprint caching [8], heuristic based approaches to group the fingerprints which are accessed together [9][10] and estimation of temporal locality of fingerprints [11] etc. Caching optimizations are also important to enhance system performance. Caching based solutions are SSDs (Solid State Drives) for metadata storage [12], SSDs as content-based data cache [13] and workload specific cache sizing between data cache and

metadata cache [14] etc. Most of them have assumed that primary workloads exhibit temporal locality in data access. Hence, they used DRAM as address-based cache with LRU policy. In cloud environment, consolidation of different workloads causes interference of workloads which affects their cached locality. In order to utilize cache space efficiently, content-based cache is required. Some cache blocks may receive high references during the initial period of stay in the cache, but may have no references in the recent interval of time. In another scenario, few blocks may get high reference count during short interval of time. While differentiating among different cache blocks with same access frequency, their staying period in cache is helpful. In order to utilize cache efficiently, there is a need to differentiate among these blocks.

The centralized deduplication storage systems can handle limited amount of data. If data being stored on these systems scales beyond the limit, system performance is degraded. In order to improve scalability and deduplication efficiency, distributed deduplication system is required for cloud environment. The performance of distributed deduplication system depends upon, how well the data is assigned to server to achieve high duplicate elimination ratio with load balancing. In the context of distributed deduplication systems, if high duplicate data elimination is targeted, all files of same type need to be sent to the same server which leads to load imbalance problem. If load balance is to be achieved, then duplicate data elimination has to be compromised. Thus achieving high amount of duplicate elimination with load balancing is a difficult task. The above mentioned challenges motivated the present work for designing of centralized or distributed primary storage deduplication systems working at block level and file level to handle primary workloads exhibiting random access patterns and weak temporal locality.

1.1.1 Aim

This dissertation aims to design and develop primary storage deduplication schemes for cloud environment.

1.1.2 Objectives

The main objectives of this dissertation are stated as follows:

- To get insight into the state-of-the-art deduplication systems.
- To design and develop centralized primary storage deduplication system at block level with similarity based indexing approach.
- To design and develop centralized primary storage deduplication system at block level with content-based data cache.
- To design and develop file semantic aware centralized primary storage deduplication system.
- To design and develop file type aware distributed deduplication system.

1.2 Overview of the contributions of this thesis

Now a days, research on primary storage deduplication systems has gained abundant attention due to the demand for efficient data storage. While designing primary storage deduplication system for cloud environment, in this thesis, main parameters considered are scope (centralized or distributed), workload locality assumption, timing (inline, offline or hybrid), level (file or block), indexing (full or partial), duplicate elimination (exact or partial) and cache (address-based or content-based).

1.2.1 Centralized primary storage deduplication system at block level with similarity based indexing approach

Primary workloads exhibit latency sensitivity, random access patterns [3] and weak temporal locality [2][15]. These features inhibit application of inline deduplication for primary storage systems on the I/O path as it incurs extra latency due to overheads of deduplication process. Deduplication is a CPU intensive process due to chunking and hashing as well as memory and I/O intensive process due to index lookup. The direct application of

deduplication on the I/O path causes an increase in latency. Deduplication metadata consists of information about fingerprints and the location of stored data blocks on the disks. Fingerprints have random values and poor locality, due to which the disk is accessed frequently. This is known as the disk-bottleneck problem [16]. As metadata access overhead is increased, system performance is affected. Performance of deduplication system can be enhanced by caching the frequently accessed metadata. But metadata accesses may exhibit random access patterns. Hence, temporal and spatial locality based caching can be ineffective. When deduplication is applied to an incoming write request, a contiguous sequence of blocks can be scattered on the disk leading to data fragmentation. Eventually, performance of I/O requests for sequential data is affected.

HDS is designed for primary workloads that exhibit random access patterns and weak temporal locality. It works at block level supporting inline as well as offline deduplication. In HDS, I/O requests are categorized as four types – small read, large read, small write and large write and are processed as per their respective priorities. In order to process the read requests, the metadata is searched to identify the corresponding physical blocks to complete the request. In case of small write requests, metadata is stored in an efficient hash table. However, in case of large write requests, appropriate segments are identified through request graph and metadata is stored in similarity based buckets. HDS leverages similarity based partial look up to enhance deduplication performance.

Main contributions of this work are:

- Combines inline as well as offline deduplication techniques to reduce latency on I/O path.
- Applies similarity based grouping of the segments to reduce the search space for duplicate identification.
- Uses locality order preserving indexing to improve the performance of sequential accesses.
- Uses selective deduplication to eliminate data fragmentation.

Performance of the HDS is compared with native system and full deduplication system

based on parameters namely (i) normalized response time in terms of number of metadata blocks accessed inline, (ii) metadata overhead, in terms of the count of metadata operations (insert, delete, update and search operations), per data block, (iii) metadata overhead, in terms of number of metadata blocks accessed, per data block, (iv) write requests eliminated, (v) average overhead per read request, (vi) average data block segment length, (vii) normalized storage optimization and (viii) read and write response time statistics. HDS is found to perform consistently better than full deduplication system and native system.

1.2.2 Centralized primary storage deduplication system at block level with content-based data cache

Deduplication system's data cache can be maintained as address-based cache or content-based cache. Existing works [11][8][14] have used address-based data cache and used LRU policy for caching. However, cloud storage is shared by different workloads which may have I/O requests with different addresses containing duplicate data. When address-based cache is used, such I/O requests result in populating the cache with duplicate data, leading to inefficient utilization of the cache. Due to consolidation of workloads on to a system, localities of workloads get interfered with each other which leads to the replacement of cached locality of one workload by other workloads. In addition to this, deduplication of data causes a single data block to be shared among multiple workloads. Such shared data blocks are accessed by different workloads. Their access frequency changes over a period of time. Thus, recency alone (Least Recently Used (LRU)) or frequency alone (Most Frequently Used (MFU) or Least Frequently Used (LFU)) based cache replacement policies are not effective in the cloud environment.

In order to overcome challenges associated with address-based cache in cloud, HDS containing the content-based cache with a new replacement policy - Modified Adaptive Replacement Cache (Modified-ARC), is proposed. In order to select the victim for cache replacement, popularity of the content, which is a combination of recency and frequency of references is considered. The main contributions of this work are given below.

- Proposes similarity based indexing and selective deduplication.

- Maintains content-based data cache.
- Uses deduplication to avoid caching of duplicate data blocks.
- Introduces data block popularity metric based on weighted frequency, idle staying period and recency.
- Proposes popularity metric based cache replacement policy to cope up with the weak temporal locality.

Experiments are conducted by varying the cache size from 10% to 80% of the total working set size for Web and Home datasets. For Mail dataset, the cache size is varied from 2% to 20% of the working set size, because of its high duplicate I/O requests and duplicate content. Metadata cache size is reserved at 4% of the total cache size, for all of the traces. Performance of proposed HDS is compared with full deduplication and native systems, with different cache replacement policies, by varying cache sizes. The parameters used for the comparison are (i) hit ratio, (ii) average read response time per 4 KB block, (iii) average write response time per 4 KB block, (iv) normalized effective writes performed, (v) metadata overhead, in terms of number of metadata blocks accessed, per data block, (vi) write requests eliminated and (vii) average overhead for read and write requests. Experimental results have shown that HDS with content-based cache enhances system performance. Effective writes performed is reduced with Modified-ARC compared to LRU. Overall, I/O system performance is improved with Modified-ARC based HDS.

1.2.3 File semantic aware centralized primary storage deduplication system

Primary storage deduplication systems work at either block level [8][14] or file level [17][18]. Block-level deduplication has only the content of the block whereas, file-level deduplication has content and file semantics such as file type, size, access time and modification time etc. In both of these approaches, deduplication can be applied directly on the I/O path of the primary storage systems. Irrespective of the level, applying deduplication raises problems such as extra latency on the I/O path, disk-bottleneck and data fragmentation. Most

of the recent research works have tried to address disk-bottleneck and data fragmentation problems at the block level. However, addressing these problems, deduplication systems that are aware of file semantics, can be found rarely. Research works on file type aware deduplication can be seen in the context of either backup deduplication [18] or distributed primary storage systems [17][19].

Primary storage deduplication systems have domination of small size file access over large size file access. Applying deduplication on small files is a resource-intensive task with less space-saving. In the context of large size files, file type is useful to determine the level of duplicate content. Based on content redundancy, files can be partitioned as high duplicate, low duplicate and unpredictable duplicate. It has been observed that data redundancy, across different types of files is negligible [5] [20]. Deduplication among files of mismatched types results in increased deduplication overhead with minimal saving in storage capacity. If deduplication metadata is maintained separately based on file size and file type, deduplication overhead can be reduced. Whole file chunking for low duplicate file and variable-size chunking for highly duplicate file reduces resource overhead and improves deduplication ratio. However, application of variable-size chunking is not feasible in the primary storage systems, due to high overhead. The file type-specific deduplication strategy helps in reducing the deduplication overhead and achieves storage space-saving.

In this work, the File Aware DeDuplication system (FADD) is proposed. Files are categorized as small files and large files. Based on extensions, large files can be categorized broadly as high duplicate (H) type, low duplicate (L) type and unpredictable duplicate (U) type. H and U type large files undergo segment level deduplication and L type files undergo file-level deduplication. Fixed-size chunking is applied for H and U type files to improve the deduplication ratio. Whereas, whole file chunking is applied for L type files which decreases usage of computational resources. Deduplication metadata of large files is maintained separately for each type and hash table is used for small files of all types, so that overall metadata overhead can be reduced.

Main contributions of this work are as follow

- Proposes file categorization based on data redundancy.

- Identifies similar segments and groups them into buckets, for large size files of H and U type .
- Applies whole file deduplication for large size files of L type.
- Organizes metadata into a hash table for all small size files efficiently.

Performance of proposed FADD is compared with HDS, full deduplication and native systems. File type aware deduplication enhances overall I/O performance and storage efficiency. In the study, the parameters - metadata overhead, overhead for inline processing of read request, the average length of stored segments, storage space-saving, average read response time, average read throughput and average write throughput are measured. File type-specific deduplication saves the resources and reduces deduplication overhead. In the experiments conducted, it is observed that the FADD system performed better than HDS, full deduplication and native systems.

1.2.4 File semantic aware distributed deduplication system

Distributed deduplication system consists of multiple data servers handling data from multiple clients simultaneously. Distributed deduplication system has to consider various issues such as the location of deduplication, splitting of deduplication tasks between client and data server, proper assignment of data and load balancing. Deduplication can be performed at client or at data server. Applying deduplication at client avoids duplicate data transfer but cannot eliminate duplicate data among clients. Whereas, deduplication at data server may require more network bandwidth, because all clients need to transfer their entire data. Data sever eliminates the duplicate data among all the clients which increases the load on data server. Apart from this, deduplication at data server has to consider whether duplicate data elimination is to be performed within a data server or across all data servers. The former approach results in information island [21] problem. The latter approach has to maintain a global index table which has to be queried by all data servers for identifying duplicate data, which results in a bottleneck. In order to distribute the deduplication processing load, deduplication tasks can be split between client and data server. Data chunking and fingerprint computation which are CPU intensive, can be performed at the client.

Whereas, index lookup and duplicate eliminations that are memory and I/O intensive tasks, can be performed at the data server. File type aware data assignment gives a better deduplication ratio while file type oblivious assignment incurs less overhead. Another type of categorization of data assignment approach is stateful approach and stateless approach. In a stateful approach, the data present at the data servers is considered while assigning data to the data servers. In the stateless approach, the decision is taken purely based on only the data being assigned. In the former approach, load balancing can be achieved, whereas it is ignored in the latter approach.

In this work, Distributed Deduplication System (DDS) is proposed to perform file type aware, hybrid i.e., stateful and stateless distributed deduplication. The main contributions of this work are as follow.

- Classifies files based on the percentage of duplicate content.
- File type-specific allocation of sets of data servers.
- Applies file type-specific deduplication strategy and routing approaches.
- Selects the suitable data server based on probabilistic estimation of duplicate content.
- Proposes similarity-based indexing at the data servers.

Experiments are conducted by varying the number of data servers from 2 to 64. In the study, the parameters namely (i) data skew, (ii) normalized space saving, (iii) read throughput, (iv) write throughput and (v) assignment time are measured. Performance of DDS is compared with extreme binning system [15] and it is found to be consistently better in achieving the load balancing with reasonable space saving.

1.3 Benchmark datasets

Prototype of the proposed systems are implemented and simulated under the Linux operating system running on Intel i7 processor based system. Trace driven experiments are conducted to assess the system performance. Traces include standard I/O traces taken from

three production systems at Florida International University (FIU) and some locally collected data sets. The following datasets are considered to evaluate the performance of the system.

- Mail [22], I/O traces consist of the I/O requests generated by the virtual machines running mail server.
- Web [22], I/O traces consist of the I/O requests generated by the virtual machines running web server.
- Home [22], I/O traces consist of the I/O requests generated by the virtual machines running NFS server.
- *Book-ppt* dataset consisting of books, documents and ppts collected from desktops of department laboratory.
- *Linux* dataset consists of a collection of Linux kernel source code with version 5.x.y.
- *Video-image* dataset consists of a collection audio, video and images from desktops of department laboratory.

The first three datasets namely Mail, Web and Home are used in assessing the performance of centralized block level deduplication systems. In order to assess the performance of file level deduplication system, five datasets are used namely Mail, Web, *Book-ppt*, *Linux* and *Video-image*. File level deduplication systems are categorizing the files based on data redundancy level. The *Book-ppt* and *Linux* datasets are categorized as H-type, Mail and Web datasets are categorized as U-type and *Video-image* dataset is categorized as L-type. For Mail and Web datasets (traces available without data), a file is identified as a sequence of read/write requests from the same process for the consecutive LBAs.

1.3.1 Performance parameters of deduplication system

The following parameters can be used to assess the performance of a deduplication system:

- **Storage optimization**: Storage optimization is measured by counting the total unique block writes issued by the processes and actually stored unique blocks after deduplication. Let T indicates total unique block writes issued and S denotes total number of unique blocks stored then

$$\text{Storage optimization} = \frac{(T - S)}{T} * 100 \tag{1.1}$$

- **Metadata overhead**: Overhead of metadata accesses is measured through two parameters: i) number of metadata blocks read/written per data block read/written. ii) count of metadata operations (insert, delete, update and search) per data block read/written.

- **Inline read overhead** is measured for read operation as the number of metadata blocks being read/written per data block on the read path.

- **Average segment length** is the average of length of deduplicated data block segments.

- **Average read response time** is the amount of time deduplication system takes to process a read request that is making a request and receiving the first data block requested.

- **Average Write response time** is the amount of time deduplication system takes to process a write request

- **Write requests eliminated** is the difference between total number of writes issued and actual number of writes performed at the disk system

- **Data skew** is defined as the ratio of storage space on the maximum loaded data server over average storage space utilized.

- **Normalized space saving**: Space saving is computed as the difference between the total amount of data to be stored and the actual amount of data stored by the data

server(s). Maximum duplicate content that can be eliminated using full deduplication approach is used to normalize the space saving.

1.4 Organization of the thesis

The rest of the chapters of this thesis are organized as follows:

Chapter 2 describes the recent state-of-the-art works on primary and backup storage systems which are further classified as centralized and distributed deduplication systems.

Chapter 3 proposes hybrid deduplication system which works at block level. This chapter aims at solving disk-bottleneck and data fragmentation problem by proposing similarity based indexing and selective deduplication.

Chapter 4 proposes hybrid deduplication system with content based cache with novel cache replacement policy Modified-ARC policy that considers various factors - weighted frequency, idle staying period and recency of data block.

Chapter 5 presents file semantic aware primary storage deduplication system. In this system, files are classified broadly into three categories based on data redundancy as high duplicate, low duplicate and unpredictable. High and unpredictable files undergo segment level deduplication and low redundant files undergoes file level deduplication. Deduplication metadata is maintained separately based on file size and type.

Chapter 6 presents file semantic aware distributed deduplication system. In this work, files are categorized as high duplicate, low duplicate and unpredictable. Highly duplicate and unpredictable files are fixed-size chunked and segments are created. Segments are grouped into superchunks. These superchunks are assigned to data servers using stateful routing approach. The best data server for assigning superchunk is selected probabilistically. Low duplicate files are whole file chunked and assigned to data servers using stateless routing approach.

Chapter 7 gives the conclusions of the thesis and future directions.

All the deduplication systems in chapter 3, 4, 5 and 6 have tried to solve problems associated with respective deduplication approach and improve system performance.

Chapter 2

Background

There is an enormous amount of data growth due to various technological trends such as Big Data, Cloud Computing and the Internet of Things. As per International Data Corporation [1] report, the amount of data will exceed 175 ZB by 2025. However, 75% of this data is duplicate. Other research investigations are also conducted to find duplicates percentage in workloads. Among them Microsoft [5][4] has observed 90% duplicate data in backup workloads and EMC [23] has observed 65% duplicate data in primary workloads. In order to avoid duplicate data storage and utilize storage space efficiently, storage optimization technique is required. There are two storage optimization techniques available in the literature, namely data compression and data deduplication. Data compression technique performs byte by byte or string of bytes comparison in the content of one or few files for identifying duplicate data. Whereas, deduplication technique computes hash value for the content of large size blocks ($\geq$ 512 bytes) and uses it for duplicate data identification among large number of files, which is a better scalable approach compared to compression. Deduplication is the most appropriate technique for large storage systems used in industry as well as academia.

In storage systems, duplicate data may exist in a file or across files. This is known as data redundancy. If the redundant data is stored in the buffer cache, it results in inefficient use of the buffer cache. Applications may generate multiple write requests with the same data, which is known as I/O redundancy. Thus, in storage systems, duplicates may exist in either files or buffer caches or in I/O requests. Broadly, research work on deduplication can

be classified as follows:

1. Data deduplication
2. Cache deduplication
3. I/O deduplication

When data deduplication is applied to primary or backup storage systems, duplicates are eliminated and unique data is stored on the disk. Thus, the elimination of duplicates allows storage of more data on the disk [16][24][25]. Traditionally, the cache is organized based on the addresses of data blocks, which may lead to have duplicate data in the buffer cache. If the cache is organized based on the content, while adding new data, duplicates can be identified and eliminated. This is called cache deduplication [26][27][28]. When multiple write requests are generated for the same data block, if writes are delayed, the data is overwritten within the cached buffer. Such multiple writes are eliminated and final write operation is issued to the disk. If multiple write requests with the same data are generated for different data blocks, physically one write request is issued to a common block on the disk and all the original data blocks are set to point to the common block. Thus, the elimination of multiple writes to the same block or the elimination of writing the same data to different blocks is called I/O deduplication [29][30][31][32][33].

2.1 Deduplication system

Deduplication process consists of five phases namely chunking, hashing, indexing and searching, duplicate elimination and storage management. Sequence of these steps are shown in Figure 2.1 and are described in detail in the following subsections.

2.1.1 Chunking

Chunking is an initial step in deduplication which divides the given file data into fixed-sized or variable-sized partitions known as chunks. There are three types of chunking namely - whole file chunking, fixed-size chunking and variable-size chunking [34][35][36][37].

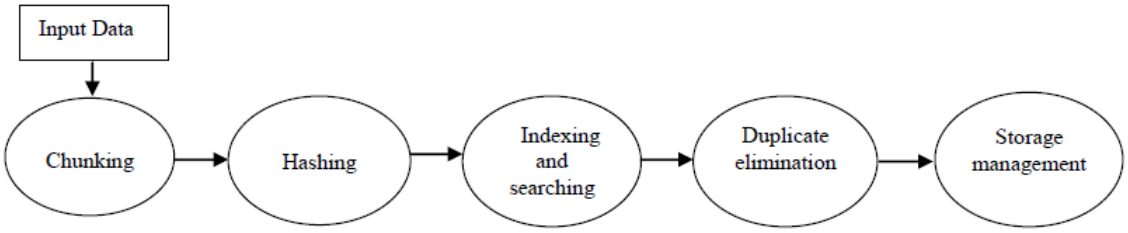


Figure 2.1: Deduplication system

Whole file chunking considers total file content as a chunk. Fixed-sized chunking divides file data into fixed-size chunks of size 4 KB, 8 KB etc. Fixed-size chunking suffers from boundary shift problem [38][37][39]. In variable-size chunking [40][41][42], file data is divided into variable-sized chunks based on some patterns and conditions. Variable-size chunking is CPU intensive process that has to keep track of content based data block boundary. The chunk size determines the total number of chunks, index table (metadata) size and index lookup time for duplicate identification and it also affects duplicate elimination ratio. The overhead incurred due to chunking can be alleviated if chunking can be performed efficiently based on data redundancy level in workloads. Primary workloads have low data redundancy, hence fixed-size chunking is preferred. Whereas, backup workloads have more data redundancy and prefer variable-size chunking.

2.1.2 Hashing

Deduplication system avoids the storage of chunks with same content. To identify the content similarity of chunks, there are two approaches namely byte by byte comparison [43][44][45] or cryptographic hash value comparison[46][47]. The former approach, being time-consuming and I/O intensive, is rarely used. Whereas in latter approach, hash values used are short to store, retrieve and compare more efficiently. Hash value or fingerprint for the content of the chunk is computed using cryptographic hash functions such as Secure Hash Algorithm (SHA-1) or Message Digest (MD-5). Size of the hash value for SHA-1 is 160 bits and for MD-5, it is 128 bits. The computation speed of MD-5 is faster than SHA-1. Cryptographic hash collision probability of two different data blocks is several orders less than the probability of disk block error [25][48].

2.1.3 Indexing and searching

Deduplication system needs to compare incoming chunk fingerprint with already stored chunks fingerprints for duplicate identification. Hence, efficient indexing of fingerprints is required. There are two approaches for indexing namely exact index and partial index. Exact index maintains fingerprints of all chunks. Partial index maintains only representative fingerprints of groups of fingerprints and each group of fingerprints is stored in some container. These groups are formed based on locality, similarity or both. The locality-based indexing approach stores all fingerprints and deduplicated data of the workload together in the same container. In the similarity-based indexing approach, similarity among data is exploited and similar data and their fingerprints are stored together. Fingerprint index should be persistent, so it is stored on hard disk. In order to determine whether the given data is duplicate, computed fingerprint of the data is searched in the index table. If fingerprint exists, that data is treated as duplicate data. Otherwise, it is treated as unique data and inserts the fingerprint in the index table.

2.1.4 Duplicate elimination

There are two types of duplicate elimination - exact and near-exact duplicate elimination. In the former approach, all duplicate data is eliminated. Whereas, in the latter approach, duplicates are eliminated selectively. If the deduplication results in data fragmentation beyond a threshold value then duplicate data is not eliminated. Conditional elimination of duplicate data is called selective deduplication.

2.1.5 Storage management

Deduplication eliminates duplicate data blocks and stores one copy which is referenced by two or more logical blocks. Hence, logical block to physical block mapping and reference count management is required. During the deduplication process, physical block's reference count may reach zero. Such physical blocks are reclaimed by the garbage collector. In addition to this, storage management optionally can perform data defragmentation also.

2.2 Deduplication system parameters

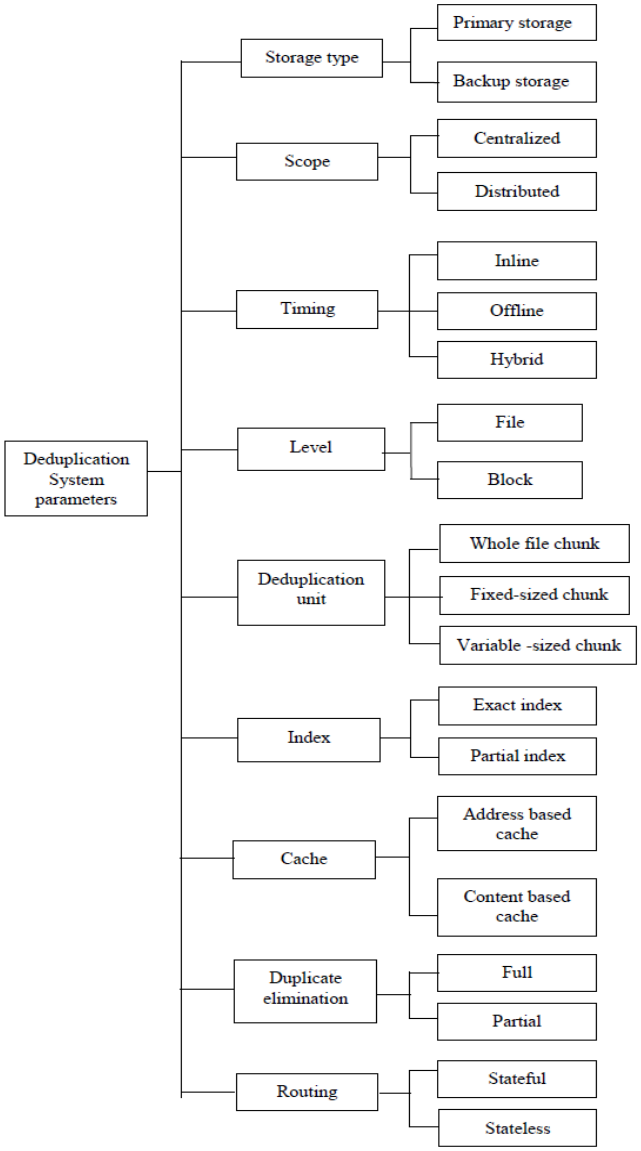


Figure 2.2: Classification parameters

Deduplication systems can be categorized based on different parameters such as storage type, scope, timing, level, deduplication unit, index, cache, duplicate elimination, routing etc. as shown in Figure 2.2.

Broadly storage systems are classified as primary and backup storage systems. Primary storage deduplication system applies the deduplication process to the online I/O path.

Backup storage deduplication applies the deduplication process while taking the file system backup.

The scope of a deduplication system determines whether deduplication is performed at a single storage node or multiple storage nodes. Deduplication at a single node is called centralized deduplication [16][25][49] and deduplication at multiple storage nodes is called distributed deduplication[50][51][52][53][54][55]. In a centralized deduplication system, all deduplication tasks are performed at a single node. It works well for small-scale data sets. In a distributed deduplication system, deduplication tasks are split between client and server nodes, which enhances the scalability of the system.

The timing of the deduplication can be either before or after storing the data. Based on the timing parameter, deduplication systems can be classified as inline, offline or hybrid deduplication systems. Inline Deduplication eliminates duplicate data before writing to the disk. In offline deduplication systems, data is written to the disk as it arrives. During idle periods, deduplication is performed. It is also known as post-processing deduplication. Hybrid deduplication system applies inline followed by offline deduplication. During the inline phase, partial deduplication is performed and skipped duplicates are eliminated during the offline phase.

The level of a deduplication determines whether deduplication should be applied at file or block level. At file level, file related semantic information such as size, type, access time, modification time etc. is available and this information can be used to avoid wastage of resources. In order to apply file level deduplication, file system needs to be modified [56][3]. Whereas in block level approach, only content is available and no other semantic information is provided. Hence block level deduplication system [8][14][2] can be plugged independent of file system.

Deduplication unit is the chunk of data, which is used as a unit to search for duplicates. It can be classified as whole file chunk, fixed-size chunk or variable-size chunk. Whole file chunking is the simplest chunking approach among all chunking methods. Fixed-size chunking divides file data into chunks of size 4 KB, 8 KB etc. Variable-size chunking is also known as content-defined chunking which partitions file data based on content.

Indexing of all fingerprints of all chunks or only representative fingerprints of groups

of chunks can be done. Former index is called as full index and the latter one is called as partial index.

Deduplication system cache can be organized as address based cache or content based cache. Content based cache [57] utilizes the buffer cache more efficiently, because it eliminates the duplicates.

Duplicate elimination type can be either full deduplication or partial deduplication. In full deduplication, all duplicates are eliminated and in partial deduplication, duplicates are eliminated selectively.

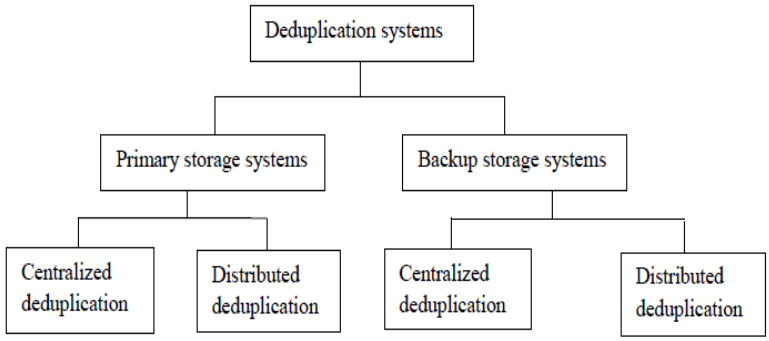
In distributed deduplication system, deduplication can be done either at the client or at the server. In order to assign / to route the data to servers, there are two approaches namely stateless and stateful routing. Stateless routing assigns the data chunk to a server based on the content of the chunk to be assigned. In this approach, load balancing issue is not considered. Stateful routing considers the content of already stored data chunks and may balance the load also.

2.3 Classification of deduplication systems

Research works in the area of deduplication can be broadly categorized based on two parameters namely storage type and scope as shown in Figure 2.3

2.3.1 Primary storage deduplication systems

Primary storage system handles latency sensitive I/O requests directly coming from the file system. Redundancy in this data is less. Primary storage deduplication can be applied to a single storage node (centralized) or to multiple storage nodes (distributed).



iDedup [8]	DEDIS [58]	DDFS [16]	Extreme Binning [15]
POD [14]	SAUD [19]	Chunkstash [12]	DEBAR [59]
iSieve [60]	D^3 [17]	Sparse Indexing [49]	MAD2 [61]
DBLK [24]		RMD[62]	HYDRAstor [53]
Dmdedup [2]		Capping [63]	Dong et. al. [64]
ProSy[65]		CBR [66]	Boafft [67]
PDFS [3]		HAR [68]	Σ-Dedup [69]
HANDS [9]		DCO [70]	PRODUCK [71]
Leach [10]		SMR [72]	SAM [18]
HPDedup [11]			DDE [73]
			Dedupv1 [74]
			SILO[75]
			AA-Dedup [20]
			AA-Plus [76]

Figure 2.3: Classification of research works in the area of deduplication systems

2.3.1.1 Centralized primary storage deduplication system

The parameters of centralized primary storage deduplication systems are given below.

1. Storage type–> Primary storage
2. Scope–> Centralized deduplication
3. Timing–> Inline, offline and hybrid
4. Deduplication unit–> Chunk and file
5. Index–> Full index and partial index

6. Duplicate elimination –> Exact and partial deduplication

Applying the deduplication technique in centralized primary storage systems for eliminating duplicate data causes the following problems.

1. Primary workloads are latency-sensitive. Applying deduplication directly on the I/O path imposes extra latency due to deduplication processing.
2. Indexed fingerprint values are stored on a hard disk. While identifying the duplicates, index lookup causes frequent accesses to the disk, which is known as the disk-bottleneck problem.
3. Deduplication eliminates duplicate blocks. Due to this, sequential data may get scattered. This is known as the data fragmentation [72][77][78] problem. Subsequently, it affects read performance.

In literature, many solutions have been proposed which can be categorized as selective deduplication, indexing optimizations, caching optimizations etc.

iDedup [8] is recognized as a state-of-the-art primary storage deduplication work. It is a file system level deduplication system. iDedup assumes the existence of spatial and temporal locality in primary workloads. The temporal locality property is used to populate the cache with fingerprints of frequently accessed files. The spatial locality of duplicate data on the disk and the incoming file can be used to deduplicate write requests and alleviate read amplification problem. iDedup ignores small file deduplication and selectively deduplicates large requests if the incoming write request has a duplicate data block sequence of minimum length three. Both of these techniques can improve the index lookup time. However, for primary workloads with random access patterns, the performance gain using the locality-based index lookup optimization is limited.

Performance-Oriented data Deduplication (POD) [14] is a block-level deduplication system. POD assumes that primary workloads exhibit locality features and has shown that the deduplication performance is more affected by small requests. Hence, it performs small as well as large file deduplication. It consists of two modules, namely Select-Dedupe and iCache. Select-Dedupe performs selective deduplication of write requests. Deduplication

of a write request is performed if it has a minimum of three consecutive duplicate blocks. In order to enhance I/O performance, an adaptive cache partition strategy is implemented in the iCache module which adjusts index cache size and data cache size depending on workloads. If write traffic is more, the size of the index cache is increased, otherwise the size of the data cache is increased.

I-Sieve [60] proposes an indexing technique for client-side deduplication. It consists of three components namely Deduplication engine, Multi-level cache and Snapshot module. The Deduplication engine performs block-level data deduplication and maintains a fingerprint table and an I/O remapping table. I/O remapping table has two fields namely Logical Block Address (LBA) and pointer to fingerprint table. The fingerprint table is structured as a multi-index structure. For this purpose, the first three 8-bits of the full fingerprint of the data block are used to create three-level index. When write requests arrive at the deduplication engine, chunking, hashing and index lookup operations are performed. Then request is transferred to disk request queues. The Multi-level cache module handles cache requests and maintains two-level caching using SSD and RAM. Complete index table and remapping table is maintained in SSD. Depending on temporal locality, RAM is populated with index data. The snapshot module is maintained for reliability of deduplicated data that takes snapshots at periodic intervals.

Deduplication BLock Device (DBLK) [24] is a block-level deduplication system aimed at reducing index lookup time on the I/O path by using a multi-layer Bloom filter. DBLK maintains hash index (fingerprint to physical Block Address (PBA) mapping) and block map (LBA to fingerprint mapping). In order to accelerate search operation on hash index, multilayer bloom filter is maintained. Availability of a fingerprint on disk is confirmed using multilevel Bloom filter. If its presence is confirmed, then the second level filter is used to find the fingerprint's location on the disk.

Dmdedup [2] is a block-level deduplication system with various metadata management techniques. The framework consists of a deduplication module, Logical Block Number (LBN) to Physical Block Number (PBN) mapping, hash index (hash to PBN mapping), space manager and chunk store. The incoming write request is given to the deduplication module where fixed size chunking, fingerprint computation and index lookup in the hash

index are performed. Mapping tables are appropriately updated. The space manager keeps the information about space on the data device and reference counts. In addition to this, the space manager is responsible for allocating new blocks and deallocating unreferenced blocks. The chunk store module is responsible for saving the data to the data device.

ProSy [65] is an inline primary storage deduplication system that deduplicates large files of size greater than 2 MB. The file is partitioned into fixed-length segments (2 MB to 8 MB) and content-defined chunking is applied on each segment to generate variable-sized chunks known as sequences. Sequence is a unit of deduplication and for each sequence rolling checksum is computed. Similar sequences are identified based on top three highest frequency rolling checksums and added to a category. For an incoming file, deduplication is performed for each segment searching in the respective categories.

For partial data lookup in quickly picked data subsets, the Partially Dedupped File System (PDFS) [3] employs a similarity-based indexing technique. It intercepts dirty blocks at the block layer and the Fletcher2 fingerprint is computed for each block. In order to create variable-sized segments, if the computed fingerprint of the data block is evenly divisible by the divisor, that block determines the segment boundary. Using Locality Sensitive Hashing (LSH), similar segments are detected and grouped into buckets. LSH is a CPU-intensive process that makes inline deduplication systems difficult to implement. In order to use inline deduplication, the write system call code path is shortened by delaying the dirtying phase. Before committing the dirty data, for deduplication parallel fingerprint computation and search is performed in the background using a write order-preserving optimization approach.

Heuristically Arranged Non-Backup In-line Deduplication System (HANDS) [9] is a block-based deduplication system. Fingerprint index mapping (a fingerprint is mapped to a chunk), index cache (which is a subset of the fingerprint index) and working set table (fingerprints are mapped to working sets of fingerprints) are the three modules that make up the framework. The neighbourhood partitioning method is used to build working sets of fingerprints statically. It accomplishes this by using the time and offset of previous I/O accesses. Working sets are written to the on-disk fingerprint index. When a write request arrives, fingerprint is calculated for the content of data block using SHA-1 hashing

algorithm and compared against the fingerprints in the index cache. If a duplicate request is found, it is marked as such and metadata is updated. Otherwise, the request is sent to the disk. The existence of a fingerprint on the disk can be verified by using a bloomfilter. If a fingerprint is found, the working set table is searched to determine which working set group it belongs to, and the index cache is loaded. Otherwise, the request is regarded as a new one and indexing information is updated.

Leach [10] maintains an in-memory fingerprint cache that is populated based on prior I/O accesses. Aside from that, the fingerprint index is stored on the disk as a Splay tree, a self-adjusting data structure that performs a move-to-root operation after each I/O access to move the requested node to the root. The upper part of the tree will have highly accessed fingerprints that can be used as a working set due to the locality of references. Working sets can be dynamically updated at regular intervals by examining previous I/O accesses. The working set change ratio at a given time is computed and compared to a threshold to control the tree's splaying. If the ratio is less than the threshold, no changes to the working sets are made. Otherwise, the tree is splayed to generate a new working set. Index lookup time is reduced and overall deduplication process time is improved.

Hybrid Prioritized Data Deduplication (HPDedup) [11] is a cloud-based deduplication mechanism for primary workloads received from many data streams generated by various clients. It assumes that the primary workloads of different clients in the cloud exhibit random access patterns. Each data stream has its temporal locality and a working set of different sizes. As a result, equally allocating cache among multiple data streams may be ineffective. Locality-based indexing must consider cache size and the number of fingerprints to be cached to increase deduplication system performance. The fingerprint locality size for each data stream is evaluated using the reservoir concept and the cache is allocated based on that estimate. The spatial locality feature of the data stream helps to alleviate the disk fragmentation problem. An incoming data block segment is deduplicated only if a predefined threshold length of the duplicate segment is found. The threshold is determined dynamically since different data streams have varying spatial localities.

2.3.1.2 Distributed primary storage deduplication system

Different possible values of the parameters of distributed primary storage deduplication systems are given below.

1. Storage type–> Primary storage
2. Scope–> Distributed deduplication
3. Timing–> Inline, offline and hybrid
4. Deduplication unit–> File, chunk and superchunk
5. Index–> Exact and partial
6. Duplicate elimination–> Full and partial
7. Routing–> Stateless and stateful routing

The following issues arise when the deduplication process is applied to the distributed primary storage systems.

1. Biased mapping of data chunks causes over utilization or under utilization of storage nodes, posing a load balancing issue.
2. Data chunk clustering has an impact on read performance since retrieving whole clustered data from a single node leads to formation of a hotspot.
3. If deduplication is performed across the storage nodes, lot of messages are exchanged in order to identify global matches and causes unacceptable overhead.
4. If deduplication is performed within individual nodes, to prevent overhead of global match query, it results in inter-node redundancy and forms information islands.

DEDIS [58] is an off-line distributed primary storage deduplication system for virtual machine images. It is an exact deduplication system and file type oblivious. At the client, there are two local modules namely interceptor and Duplicate Finder / Garbage Collector. In order to locate information for logical volumes of networked disks, distributed coordination and configuration service is assumed. This service has two distributed modules namely extent server and Distributed Duplicates Index (DDI). Interceptor converts logical addresses

of VMs to physical addresses. A physical block that is shared by multiple logical addresses is marked as Copy-On-Write (COW) block. When an update request for the COW block is received, a new copy of the block is created and the content is updated. Interceptor maintains two types of queues - unreferenced queue and dirty queue. Unreferenced queue has physical addresses of copied blocks. Dirty queue maintains logical addresses that are modified and consequently marked for the sharing process. Extent server is in-charge of assigning new blocks for storage and writing COW blocks. Duplicate Finder is responsible for marking aliased blocks as COW blocks. DDI is an index structure that stores information about a block's fingerprint, its physical address and the number of logical addresses pointing to that block. Garbage collector is responsible for reclaiming the space of the unreferenced blocks. Virtual machine storage writes are intercepted by the interceptor and delivered to an appropriate storage server. DDI identifies COW blocks and performs asynchronous offline deduplication of data blocks. Batch processing and in-memory cache reduce deduplication overhead.

Semantics-Aware and Utility-Driven (SAUD) deduplication [19] is an offline distributed primary storage deduplication framework that consists of a workload monitor, global file hash store, Multi-dimensional Priority-based Division (MPD) module, Utility Based Ranking (UBR) module and deduplication executor. Periodically, the workload monitor analyses the real time workload and system status. The proxy server performs file chunking and fingerprint computation and forwards this information to MPD module. For each file based on the attributes - last access time, size and type, the MPD module assigns a numeric value. Files whose last access time is less than one hour or size is less than one KB or type is encrypted are assigned priority value -1. Such files are not considered for deduplication. For other files, priority value ranges from 1 to 7 based on the last access time and increase in file size. For a given file type depending on its data redundancy level, priority value in the range 1 to 7 is assigned. MPD determines the deduplication priority of files based on these values and generates a list of deduplicatable files. The UBR module uses statistics from the workload monitor (average read latency and current deduplication ratio) and user requirements for storage system (minimal read latency and maximum deduplication ratio) and performs ranking of the files. Deduplication executor module receives dedupli-

catable file list from MPD module and the ranking of candidate files from UBR module and performs selective deduplication.

Dynamic Dual-phase Deduplication (D^3) [17] is a distributed primary storage deduplication system that performs inline as well as offline deduplication at the file level and chunk level. This system consists of a set of client nodes, a coordinator and a set of storage nodes. Depending upon the percentage of duplicate content, files are differentiated as P-type (poorly deduplicated), H-type (highly deduplicated) and U-type (unpredictable). The entire data of P-type files is treated as one chunk, whereas for H-type and U-type files the data is divided into fixed-size chunks. The data chunks generated at client nodes are sent to the storage nodes through the coordinator along with the corresponding fingerprint values. Based on the fingerprint value, the coordinator routes the data to an appropriate storage node. The coordinator maintains a similarity-based index table for global deduplication. Periodically, the coordinator computes the threshold deduplication segment length and passes it to the storage nodes. The storage nodes perform inline local deduplication of the received files that have duplicate segments longer than or equal to a threshold value. Other files that have duplicate segments shorter than threshold value are deduplicated offline on a priority basis.

2.3.2 Backup storage deduplication system

A backup is a copy of data that is taken and stored elsewhere so that it can be restored if it is lost. Full backup, differential backup, and incremental backup are the three basic types of backup. A full backup is a backup of entire snapshot of the data. Differential backup duplicates files that have changed since the last full backup. Only the changes since the last backup are backed up in incremental backups. Data duplication is the best storage optimization technique for backup storage, with 80% to 90% of data being duplicate.

2.3.2.1 Centralized backup storage deduplication system

The parameters of centralized backup storage deduplication systems are given below.

1. Storage type–> Backup storage

2. Scope–> Centralized
3. Timing–> Inline, offline and hybrid
4. Deduplication unit–> Fixed size chunk and variable size chunk
5. Index–> Exact and partial
6. Duplicate elimination–> Full and partial

The following are the challenges in implementing centralized backup deduplication system

1. Data in the incremental backup is scattered due to backing up of only the changes made since the last backup and also due to the elimination of duplicates. This data fragmentation leads to slow restoration.
2. Backup rewriting is a solution for reducing the data fragmentation in recent backups. This leads to formation of sparse containers. Reading a container which has partial valid data decreases the effective read throughput.
3. Garbage collection for reclaiming the space in sparse containers is a time-consuming process.
4. Usually the backup storage capacity is in the order of petabytes. Deduplication index must be organized using scalable structures to handle huge amounts of backup storage.

In literature, many solutions have been proposed which can be categorized as indexing optimizations and container rewriting optimizations.

Data Domain File System (DDFS) [16] tries to solve the disk-bottleneck problem. It chunks a file into variable-size segments (with an average segment size of 8 KB) based on content and compute a fingerprint for each segment using MD-5 algorithm. Segments are stored in a container. Container is partitioned into two parts - metadata and data. The metadata portion contains segment descriptors, whereas the data section contains segments. DDFS also keeps metadata such as file to segment descriptor mapping and segment index (segment descriptor to container-id mapping). DDFS has proposed summary vector, Locality Preserved Caching (LPC) of metadata and Stream-Informed Segment Layout (SISL).

Summary vector is an in-memory compact representation of the segment index. In SISL approach, segments of a stream are written to a container preserving spatial locality which improves restoration performance. Segment descriptors are also stored in the same order. Fingerprint index metadata is cached using LPC method which improves hit ratio compared to the caching based on fingerprint values. During the deduplication of segments, the segment cache is searched for fingerprints. If found, the segment is not stored. Otherwise, summary vector is searched to decide its availability on the disk. If the fingerprint is found in on-disk index, the segment is eliminated.

Chunk Metadata **St**ore on Fla**sh** (ChunkStash) [12] is flash assisted inline exact deduplication system. It consists of flash memory, RAM hash table index, RAM write buffer and RAM chunk metadata cache. Flash memory stores metadata (chunk id, metadata) of all chunks that are stored on the disk. RAM hash table index is used to index chunk id and metadata information on flash and hash collisions are resolved using cuckoo hashing. RAM chunk metadata cache is used to cache metadata of all chunks in the respective container. RAM write buffer caches chunk metadata information for the currently open container. A container is made up of 1024 chunks, each of which is 8 KB in size. When the container is filled with 1024 chunks, its metadata is written to flash and the container is sent to the disk. ChunkStash applies Rabin fingerprint based chunking on the data stream to generate chunks of average size 8 KB and SHA-1 algorithm is applied to compute its fingerprint. The fingerprint is searched in RAM chunk metadata cache. If it is not found, then search in RAM write buffer. In case of a miss, the RAM hash table is searched. If fingerprint is not found then the chunk is added to container and metadata is added to write buffer. Otherwise, find the location of chunk metadata on flash and fetch the metadata of all chunks in that container to RAM chunk metadata cache.

Sparse Indexing [49] is a near exact deduplication system. The data stream is partitioned into variable length chunks using the Two-Threshold Two-Divisor (TTTD) [79] chunking algorithm. The resultant chunk sequence is broken into segments using content-based segmentation. Each segment has a manifest that consists of information such as the hash values of its chunks, their position on the disk and the segment length. Segment manifests are indexed using chunk hashes of a segment whose first n-bits are zero. The sparse

index is an in-memory data structure that keeps sample hash value and pointer to segment manifest. For deduplicating a segment, similar segments that are stored on the disk have to be identified. Chunk hashes of an incoming segment whose first *n*-bits are zero are chosen as the sample. It searches the sparse index for matched entries and loads the respective segment manifests. One of the best manifests for deduplicating incoming segments is chosen based on the maximum number of chunks matched.

Resemblance and Mergence based Deduplication (RMD)[62] solves the fingerprint lookup problem. RMD structure consists of components namely- Fingerprint Bins (FpBin), Bin Address Tables (BATable), Dynamic Bloom Filter Array (DBA) and Fingerprint Bin Buffer (FpBinBuffer). The data stream is partitioned into variable-sized chunks using content-based chunking and fingerprint is computed for each chunk using SHA-1 algorithm forming fingerprint list. A fingerprint list is divided into a sequence of segments with each segment consisting of equal number of fingerprints. Similar segments are identified using Broder's theorem [80] and those are stored into FpBin. These segments are sorted based on fingerprint frequency. The highly frequent fingerprint is taken as a representative fingerprint (RF) of FpBin. Due to limited capacity of FpBin, less frequent fingerprints are dropped off. Fingerprint bins with similar RF form a group of fingerprint bins. BATable is a mapping table composed of RFs and FpBin addresses. The Dynamic Bloomfilter Array (DBA) is an array of bloom filters. Each bloom filter is associated with subset of RFs. RMD employs FpBinBuffer buffer that contains part of fingerprint bins for fingerprint comparisons. In order to perform deduplication, existence of RF of a segment in BATable is confirmed using DBA. When a RF is found in the BATable, the corresponding fingerprint bin is fetched into the FpBinBuffer. Otherwise, RF entry is made in BATable and new fingerprint bin is created and entry is added in FpBinBuffer.

Capping [63] divides the backup stream into 4 KB chunks. Further 5000 number of 4 KB chunks are grouped to create fixed-sized segment of size 20 MB. In order to identify fragmented chunks in segments, segments are stored in a write buffer of size of 10-20 MB. Capping proposes two approaches - container capping to avoid chunk fragmentation and forward assembly area to improve restore speed. Container capping imposes condition on the maximum number of containers (say T) a segment can refer to. The top T containers

that contain the most duplicated chunks of the target data segment are deduplicated. If a new segment refers to M containers and M >T, the chunks in $M - T$ containers that include the segment's fewest portions are rewritten and placed in new containers. In forward assembly area technique, one chunk-container sized I/O buffer and recipe buffer is used. Former buffer assembles the next M-bytes of the restored backup and the latter holds recipe of the chunk being assembled. Next, chunks are identified to fill forward assembly area. This operation is repeated to locate the first unfilled chunk position in the assembly area, load the appropriate container into the I/O buffer and then fill all parts of the assembly area that require chunks from that container.

Context-Based Rewriting (CBR) [66] divides the backup stream into blocks of size 8 KB using Rabin fingerprinting and read in a fixed-sized buffer (size 5 MB). The content in this buffer is known as stream context and duplicate block is known as decision block. A duplicate decision block is used to determine whether to deduplicate or rewrite a block in buffer. For this purpose, the rewrite utility parameter is used. Rewrite utility determines chunk fragmentation depending on the stream context and the disk context. The disk context is defined as the sequence of blocks immediately after the decision block on the disk. Size of the intersection of the stream and disk contexts, is divided by the size of the disk context, which is used to calculate rewrite utility. A chunk is fragmented if its rewrite utility is greater than the predefined minimal rewrite utility. To increase restore speed, CBR applies a rewriting limit to prevent too many rewrites.

History Aware Rewriting (HAR) [68] aims to solve chunk fragmentation problem. Their observation shows that containers with chunks that have small reference counts in the backup will continue to remain the same in successive backups. In order to identify such type of containers, a utilization parameter defined as the amount of chunks referenced by the backup, is used. There are two sorts of containers based on this - sparse containers and out-of-order containers. Containers having infrequently accessed chunks are known as sparse containers. Out-of-order containers have chunks that are accessed on an intermittent basis, causing read amplification issues. HAR recognizes sparse containers during backup and the following backup process uses this knowledge to rewrite chunks in sparse containers to improve data locality.

Delayed Container Organization (DCO) [70] framework consists of three modules Non-Volatile Memory (NVM) data management, deduplication bypass and data restore process. DCO partitions the data stream into fixed-sized segments of size 20 MB which consists of a chunk sequence. NVM data management is a multi-queue structure. Each queue stores a set of data chunks of either a segment or an intersection of segments. Apart from this, it maintains a hash table for retrieving data chunks based on fingerprints. Deduplication bypass module is responsible for calculating Usable Data Ratio (UDR) of containers that are used for deduplicating the current segment. The amount of useful data in the container, required to restore the buffer, is specified as UDR. Container construction is determined by the container's UDR. The containers with high UDR are constructed from data chunks in NVM when NVM space is used beyond the specified threshold. While restoring a target data segment, initial step is to query the hash table of the NVM data chunks. Then, as many as possible, chunks are read from NVM device.

The Sub-modular Maximization Rewriting scheme (SMR) [72] divides incoming data streams into segments made up of continuous chunks. The choice to deduplicate a segment's chunk is based on the proper selection of containers with duplicate chunks. SMR tries to choose a limited number of containers that provide more distinct referenced chunks for backup. More chunks in the backup stream can be deduplicated if more unique referenced chunks are provided for the current backup stream, saving storage space. SMR also decreases the number of disk requests made during the restore for redundant and unreferenced chunks. As a result, SMR improves recovery performance as well as the deduplication ratio.

2.3.2.2 Distributed backup storage deduplication system

The amount of data that centralized deduplication storage systems can manage is restricted. When the amount of data kept on these systems exceeds the limit, the system's performance deteriorates. For cloud environments, a distributed deduplication system is required to improve scalability and deduplication efficiency. The performance of distributed deduplication system is decided by fingerprint lookup and scalability. Fingerprint lookup performance is dependent on the fast determination of the membership of incoming data and

the identification of existing duplicate data. Scalability is decided by efficient partitioning of data into similar and dissimilar groups.

Different possible values of the parameters of distributed backup storage deduplication systems are given below.

1. Storage type–>Backup storage
2. Scope–> Distributed deduplication
3. Timing–> Inline, offline and hybrid
4. Deduplication unit–> File, chunk and superchunk
5. Index–> Exact and partial
6. Duplicate elimination–> Full and partial
7. Routing–> Stateless and stateful routing

Applying the deduplication process in distributed backup storage [81][82] [83] systems raises the following problems.

1. Improper partitioning of deduplication task between client and server causes duplicate data transfer and increase in deduplication overhead.
2. Data assignment to servers without proper mechanism causes load imbalance.
3. Deduplication within individual nodes without considering duplicates on other servers leads to inter-node redundancy.

Extreme Binning [15] divides the file into variable-sized chunks using Rabin fingerprinting technique and hash value is computed using MD-5 or SHA-1. Two-tier indexing is proposed with a primary index consisting of the representative fingerprint and a secondary index consisting of the remaining fingerprints stored in bins. Files are assigned to a single backup node for deduplication based on the file's primary index using a stateless routing approach. Because of the one-file-one-backup-node distribution, maximum parallelization can be achieved. Backup nodes can be added to enhance throughput because there is no dependency between bins. Bin and chunk redistribution is straightforward to do.

DE-duplication storage architecture for Backup and ARchiving (DEBAR) [59] is a two-phase deduplication system made up of backup clients, backup servers, a director and chunk repository clusters. A backup client divides the file into variable-sized chunks utilizing content-defined chunking and calculates SHA-1 hash values for each chunk. Backup servers maintain a file index and disk index. File index specifies fingerprints of chunks of file. Disk index is a hash table with each slot representing a bucket. The first n-bits of the fingerprint are used to identify the bucket. In each bucket entry, the fingerprint and container identifier information is stored. Each backup server has the chunk repository which is a cluster of storage nodes. Director performs job scheduling, metadata management and load balancing. The first phase deduplication is carried out at backup servers. In this phase, the fingerprint list for the file is sent to the backup server. For deduplicating chunks, the in-memory fingerprint cache is searched. Backup server determine which chunks need to be backed up and which chunks need to be considered for second phase deduplication. The file index is sent to the director. The second phase deduplication is commenced by the director and implemented by the chunk repository modules of all the backup servers simultaneously. The first w-bits of the fingerprint are utilized to identify the backup server. Disk index lookup is performed to identify new chunks and disk index updates are carried out.

MAD2 [61] is an exact deduplication system that consists of backup servers, storage proxies, metadata servers and Distributed Hash Table (DHT) based clustered storage nodes. The backup server sends file content to proxy server and metadata to metadata server. At proxy server, content is chunked using Rabin fingerprinting algorithm. File hash and chunk hashes are computed and file recipe is generated. File hash, file recipe, chunk hashes and chunk content are distributed among storage nodes. At each storage node, Hash Bucket Matrices (HBMs) to organize file hash and chunk hashes. Based on hash prefix, hash space is partitioned into n number of super buckets each of which consists of buckets of equal capacity. Buckets at same logical row of different super buckets form tanker. Each tanker has Bloom Filter (BF) to query fingerprint membership. During backup, BF is queried to identify whether a incoming hash is new or not. If query returns positive value, then hash duplicate may be present in the corresponding tanker. Next, the target bucket is determined

using the hash prefix. MAD2 performs file-level and chunk-level deduplication with HBM. It uses a DHT-based load balancing technique for the even distribution of data.

HYDRAstor [53] is a near exact deduplication system which consists of storage servers and proxy servers. Storage servers known as supernodes are organized in an overlay network as a distributed hash table and the prefix of the hash key acts as the representative of supernode. Each supernode consists of fixed number of physical nodes. A proxy provides services like locating the storage nodes, routing and caching. When a write request is received, SHA-1 hashing is applied to data block content to generate a fingerprint. The data is routed to the supernode with the matching fingerprint prefix. As data blocks are assigned based on fingerprint prefix, deduplication can be performed in parallel.

Dong et. al. [64] have proposed cluster deduplication system that divides incoming data stream into variable-sized chunks using content-defined chunking and fingerprint is computed for each chunk. Superchunk (size 1 MB) is created by storing consecutive chunks (size 8 KB) together. Minimum hash among all hashes of chunks in corresponding superchunk is taken as representative of superchunk. A superchunk can be routed by stateless or stateful approach. Superchunk is routed based on the minimum hash using stateless routing. Whereas, stateful routing routes superchunk based on already stored data and it involves communication overhead. Stateful routing sends superchunk to all deduplication nodes. Each node maintains bloom-filter to record the number of fingerprints matched. The weighted vote is computed based on the node's storage utilization. A highly weighted vote node is selected for data storage. In the context of overloaded node, superchunk is assigned using stateless routing

Boafft (Birds of a feather flock together) [67] is a distributed deduplication system for cloud storage systems. The architecture consists of clients, metadata servers and data servers. The client applies content-defined chunking on the write data stream and computes the hash for each chunk. The chunk sequence is further partitioned into segments. The minimum hash among all hashes of the segment is taken as representative of the segment. Segments are grouped to form superblocks of size varying from 4 MB to 16 MB. Thus, all representative fingerprints of the superblock are sampled to generate the feature fingerprint of the superblock. To select a data server for deduplication of superblock, the client sends a

feature fingerprint to the metadata server to know the data server address. Metadata server applies local similarity algorithm to select best data server and sends the address to the client. Client forwards the superblock to the respective data server where deduplication is applied. Data server maintains similarity-based indexing table and locality-based caching of hot fingerprints to accelerate index lookup for local deduplication.

Σ-Dedup [69] is an inline cluster deduplication framework for Big Data. Client performs chunking of data streams using fixed-sized or variable-sized chunking and computes fingerprint for each chunk using SHA-1 algorithm. Sequence of chunks are grouped into superchunk. A small group of representative hashes from a superchunk is known as a handprint. The handprint helps to determine the best data server for data storage with less communication overhead. Client sends handprint to deduplication cluster server to determine duplicate chunks. It employs a similarity hash index, which holds mappings from hashes in handprints to storage containers, to leverage locality in the data stream. Prefetching of the whole container into the cache, improves the cache hit ratio and accelerates chunk index lookup. After knowing duplicate chunks, client transfers unique data chunks for storage.

PRODUCK [71] is cluster-based probabilistic deduplication system. It consists of client application, coordinator and a set of storage nodes. Users can interact with backup system through client application. Client applies content-based chunking on file data. For each chunk, hash value is computed. Superchunk of approximately 16 MB is created by adding sequence of chunks. Coordinator is responsible for selecting the best storage node for storing the superchunk. Among storage nodes, one storage node is selected as responsible storage node which keeps track of the list of storage nodes that are storing superchunks of file. The client sends the file's identifier which is the SHA-1 hash of the file content, the number of superchunks in the file, their checksums and the size of the file to the coordinator. The coordinator selects one responsible storage node and informs the client. The client computes a bitmap vector for superchunk and sends it to the coordinator. In order to select the best storage node for storing the superchunk, the coordinator estimates the cardinality of the intersection between the chunks stored by storage node and the content of superchunk for each storage node. Next, storage nodes are ranked based on their overlap and top storage node is selected for storing the superchunk.

Semantic-Aware Multi-tiered framework (SAM) [18] is a source deduplication that exploits file semantics as a sieve during different phases of deduplication. File semantics such as locality, size, type and time stamp are utilized for enhancing index lookup during the deduplication process. It consists of three systems: client, master server and storage server. The client is responsible for separating unchanged files based on timestamp and also for maintaining two-level index. Based on the minhash value, the representative of the file, similar files are identified. After this, duplicate chunks across similar files are eliminated. Apart from this, compressed files and small files are also filtered out. The remaining files are given to the master server. At the master server, global file-level deduplication is performed.

Duplicate Data Elimination (DDE) [73] is a block-level deduplication method for the Storage Area Network (SAN) file system. In this, data and metadata are maintained separately. The client uses the SHA-1 hash function to compute the hash value for the content of data block. Data is sent to the data server for storage purpose and hash is sent to the metadata server. Data server applies three strategies to avoid duplicate data storage such as content-based hashing, COW and lazy update. Data server merges all blocks with the same fingerprint, updates block allocation map. A reference count for each block is maintained and the reclamation of unreferenced blocks is delayed. If the content of the shared block is updated, then the original block is not reclaimed. COW is applied, where a new block is allocated and the content of the block to be modified is copied and updated. Thus, it avoids duplicate data storage.

Dedupv1 [74] is a Small Computer System Interface (SCSI) based deduplication system. It consists of chunking and fingerprint component, filter chain, storage and chunk index. The chunking component splits write requests into chunks whose size varies from 4 KB to 32 KB, using content-defined chunking. Hash is computed for each chunk using the SHA-1 or SHA-256 hashing algorithm. Computed fingerprint goes through filter chain module. Filter chain consists of chunk index filter, byte compare filter, block index filter and bloom filter. It returns one of the results such as existing, strong-maybe, weak-maybe or non-existing. Existing means hash already exists, strong-maybe and weak-maybe means further hash comparisons are required, non-existing fingerprint means the

hash is new. Computed fingerprint goes sequentially through bloom filter, followed by block index filter and chunk index filter. Finally fingerprint goes through byte compare filter. If fingerprint entry is not found at any filter, the chunk will be treated as new. If an entry exists it has to be verified by going through subsequent filters. Finally, at the byte compare filter, the chunk is existing or not is decided. If exists, the chunk won't be stored otherwise, chunk is stored in the currently open container.

Xia et.al. [75] have proposed Similarity-Locality (SiLo) based near-exact inline distributed deduplication system. It consists of File Daemon (FD), Storage Server (SS), Backup Server (BS), Deduplication Server (DS), Metadata Server (MDS) which are distributed in the data centers. FD collects backup data which consists of small files ($\leq$8 KB) and large files. FD performs fixed size chunking and fingerprint computation on these files. Next, sequential chunks are grouped to create segment of size 2 MB and representative fingerprint is computed for each segment. Small files under same directory are considered as correlated files and their fingerprint set is grouped into segments. Whereas, fingerprint set of large files is partitioned into many independent segments. Sequential segments are grouped into a block of size 256 MB and representative fingerprint of block is computed. Each block has its own Locality Hash Table (LHTable). These blocks are given to MDS. MDS consists of BS and DS. BS maintains metadata information of all backup files. DS consists of Similarity Hash Table (SHTable) for similar segments, LHTable, write buffer and read cache. DS is responsible to store and lookup of fingerprints of files and chunks. In order to perform backup, FD assigns data blocks to MDS based on representative fingerprint of block using stateless routing. At DS, SHTable is searched and if a duplicate is found, whole block is treated as duplicate and metadata is updated. Otherwise, LHTable is searched and segment is written to write buffer and SHTable is updated.

AA-Dedup [20] is an application-aware backup deduplication system. AA-Dedup filters out files less than 10 KB size and the remaining files are classified into three categories as compressed files, dynamic uncompressed files and static uncompressed files. Compressed files are whole file chunked due to low data redundancy and an extended 12 Byte Rabin hash value is employed as fingerprint. Dynamic uncompressed files which include documents, PowerPoint presentations and text files are chunked using content de-

fined chunking algorithm and hash values are computed using the SHA-1 algorithm. Static uncompressed files such as virtual images, executable files and portable document files are fixed size chunked and fingerprints are computed using MD-5 hash algorithm. Hash index is split based on chunking. For each backup stream, one container is maintained. All small files are stored in the container. To maintain data locality, sequential writes are performed to open containers. When the container fills up to a predefined size, then it is written to disk. In another context, if any container has to be written to disk before filling, padding is performed before writing to the disk.

AA-Plus [76] works at file level. The hash index is split into different groups depending on application type. Fingerprints of the same application are stored together in a group handling that type. The data chunks belonging to the same application are stored together in a container, to improve spatial locality. When a write request arrives, it is partitioned into fixed-sized chunks and a fingerprint is computed for each chunk. To perform index lookup, a respective group of the index is loaded in the main memory. After identifying and eliminating duplicate data chunks, unique chunks are stored in an application-specific container.

Apart from these works, recent works on deduplication systems can be seen in the context of cloud storage [84][85][86], flash storage [87][84][83], reliability [88], caching[89], security [90].

2.4 Summary

Overview of broad classification of research work on deduplication, different phases of deduplication process and various deduplication system parameters is given in this chapter. Challenges associated with different deduplication systems, which are classified based on storage type and scope, are discussed. In order to tackle those challenges, different research works have proposed many optimizations. In this chapter, based on the applied optimizations, research works are reviewed.

Chapter 3

Hybrid Deduplication System - a block level similarity based approach

In this chapter design and developement of centralized primary storage deduplication system at block level. Primary storage deduplication systems suffers from disk-bottleneck and data fragmentation problems. In order to address these challenges, Hybrid Deduplication System (HDS) is proposed. HDS uses similarity based bucket indexing and applies separate deduplication approaches for small size and large size write request deduplication.

The advent of technologies such as Big Data, Cloud Computing and Internet of Things has given rise to aggressive digital data generation. International Data Corporation report [1], has revealed that the digital data generated will exceed 175 ZB by 2025. An alarming fact is that out of this huge data, 75% will be duplicate. Microsoft [5][6] and EMC^2 [23] have conducted studies on workload to identify percentage of data redundancy. Their studies observed that 50% data of primary workloads and 90% data of the backup workloads is duplicate. There is a need for developing mass storage systems with nil or least amount of duplicate data in the storage systems and associated methods to handle repeated as well as random I/O requests. Hence, data deduplication technique has become the most sought after storage optimization technique.

Primary workloads exhibit latency sensitivity, random access patterns [3] and weak temporal locality [2][15]. These features hinder application of inline deduplication for primary storage systems on the I/O path as it incurs extra latency due to chunking, fingerprint

computation, indexing and index look up for identification of duplicate blocks. Apart from this, application of deduplication to primary storage systems can lead to disk-bottleneck and data fragmentation problems. As the metadata of deduplication system is persistent, it needs to be stored on disk and as a consequence it has to be frequently accessed to identify duplicates and this leads to disk-bottleneck. Performance of deduplication system can be enhanced by caching the frequently accessed metadata. But metadata access can exhibit random access patterns. Hence, temporal and spatial locality based caching can be ineffective. When deduplication is applied to an incoming write request, a contiguous sequence of blocks can be scattered on the disk leading to data fragmentation. Eventually, performance of I/O requests for sequential data is affected.

Existing works on primary storage deduplication systems have tried to solve disk-bottleneck problem and data fragmentation problem using full deduplication or partial deduplication. In both the cases, metadata indexing and look-up needs to be done efficiently to enhance the performance of deduplication systems. These deduplication systems are based on either locality or similarity of the data. The locality based deduplication systems are implemented at file system level or at file system independent (back end) level. The similarity based deduplication systems are implemented at file system dependent level only. However, there is rare work found on block level similarity based primary storage deduplication system. This has motivated us to design HDS, a block level similarity based primary storage deduplication system.

Many researchers Shemi et al. [5], Meyer et al. [6] and Jin et al. [91] have observed that primary storage systems have domination of small size file accesses over large size file accesses. Most of the works [8][18] have considered that deduplication of small requests is a resource-intensive task with less space-saving. Hence, they have encouraged large size request deduplication. However, Bo Mao et.al. [14] have found that though small file deduplication results in less space-saving, system performance can be improved.

HDS is designed for primary workloads that exhibit random access patterns and weak temporal locality. It works at block level supporting inline as well as offline deduplication. HDS leverages similarity based partial look up to enhance deduplication performance. Main contributions of this chapter are:

- Combines inline as well as offline deduplication techniques to reduce latency on I/O path.
- Applies similarity based grouping of the segments (sequence of chunks) to reduce the search space for duplicate identification.
- Uses locality order preserving indexing to improve performance of sequential accesses.
- Uses selective deduplication to eliminate data fragmentation.

Rest of the chapter is organized as follows. Section 3.1 gives detailed explanation on design and implementation of HDS. Experimental results and evaluation are presented in Section 3.2. Finally Section 3.3 summarizes the work.

3.1 Design of hybrid deduplication system

This section gives detailed description about the design of HDS. Functional modules of HDS are shown in Figure 3.1.

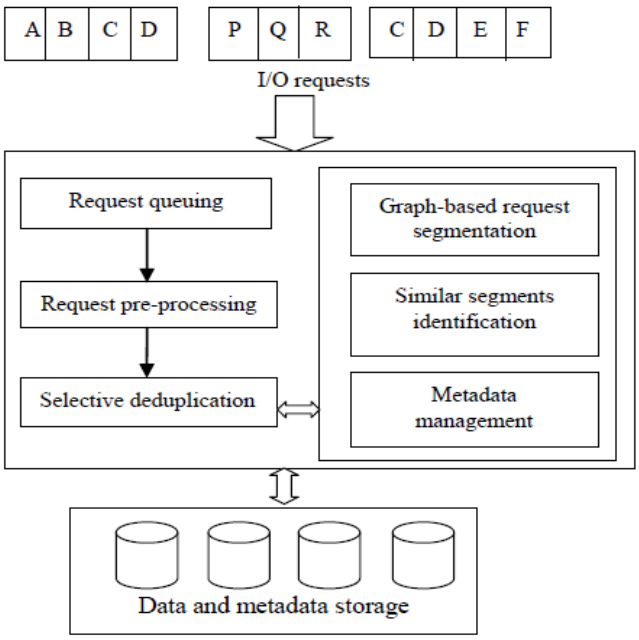


Figure 3.1: Hybrid deduplication system

It consists of request queuing, request pre-processing, metadata structures, selective deduplication, graph-based request segmentation and similar segments identification. Request queuing module is responsible for categorizing the received I/O requests as small read (SR), small write (SW), large read (LR) and large write (LW) and inserting them into appropriate queues. Request pre-processing module performs chunking and computation of fingerprints for the blocks of write requests. Metadata structures module is responsible for maintaining all metadata related to the stored data blocks. This includes hash table for small write requests and similarity based buckets for large write requests. Selective deduplication module is responsible for applying deduplication on write request selectively. For selected write request, segments of blocks are identified using graph-based segmentation submodule. For smaller segments (small requests) hash table data structure is used for quick processing. For large segments, based on similarity of segments, similarity based bucket index is searched and the corresponding bucket is identified. Bucket is searched for the duplicate blocks and such duplicates are eliminated selectively without causing fragmentation beyond certain threshold.

Algorithm 3.1 gives high level workflow of HDS. A sequence of requests are queued into four queues namely SR, LR, SW, LW. At a time one request is selected from high priority queue. If selected request is a read request, the metadata is searched to identify the corresponding physical blocks to complete the request. Otherwise, if selected request is either small or large write, separate deduplication process is applied on each type.

3.1.1 Request queuing

First step in HDS is queuing of the I/O requests. Primary storage systems receive read and write requests generally in random order. A request consists of contiguous sequence of Logical Block Addresses (LBAs). A write request includes additionally the associated data. Depending on the size, a request can be classified as a small request (Normally ≤ 8 KB) or as a large request. The performance of any deduplication system is mostly affected by small requests and the storage capacity is affected by large requests [8]. Read requests are processed inline, whereas write requests are delayed.

Algorithm 3.1 Workflow of HDS

Input: Sequence of requests is given in SR, LR, SW, LW queues

```
1:  while true do
2:      Select request r from high priority queue.
3:      if r ∈ SR ∪ LR then
4:          Perform read request processing (r).
5:      end if
6:      if r ∈ SW then
7:          Perform small write request deduplication (r).
8:      end if
9:      if r ∈ LW then
10:         Perform large write request deduplication (r).
11:     end if
12: end while
```

Based on size and type, the requests are queued into one of the following four queues. The requests in these queues are processed in decreasing order of priorities.

- Small read requests (SR)
- Small write requests (SW)
- Large read requests (LR)
- Large write requests (LW)

One request at a time is selected from a non-empty high priority queue for processing. In order to avoid the starvation of the requests in the low priority queues, priority of the older requests is increased.

3.1.2 Request pre-processing

The read requests queued are serviced inline. As part of deduplication, delayed write requests are preprocessed in the background. In the preprocessing the sub-tasks performed are - chunking, fingerprint computation and Representative IDentifier (RID) computation. Fixed-size (block size) chunking is used in the present work as it is applied at the back end.

In order to check whether the incoming chunk is a duplicate, the chunk in consideration is to be compared with all the existing chunks byte by byte. However, from the efficiency point of view, a fingerprint is computed for the chunk content using cryptographic hash algorithm MD-5 and compared against the stored fingerprints. Cryptographic hash functions are highly collision resistant and are used in standard deduplication systems for identification of duplicate data. Metadata of similar segments is stored together in a bucket. RID of the segment, which is the minimum fingerprint among all the fingerprints of the segment, is computed and is used to identify the bucket of similar segments.

3.1.3 Graph based request segmentation

After preprocessing of write requests, next step is to identify the data block segments on which deduplication can be applied. Graph based request segmentation is performed to identify the segments. Using the sequence of requests, a segment graph is constructed. A segment of contiguous sequence of logical blocks with the attributes time stamp value and reference count is considered as a vertex in the graph. If there are two consecutive requests from a process, the logical block segments represented by the vertices u and v in order, then a directed edge from u to v, is added to the segment graph. The segment graph is constructed incrementally based on the sequence of requests. From any process when a new request is received, the segment can be (i) new or (ii) existing fully or (iii) existing partially or (iv) part of an existing vertex in the graph. If it is new, a new vertex is created. If it is fully existing, the attributes such as time stamp and reference count are updated. If it is partially existing, a new vertex along with the edge(s), as per the request order, are added corresponding to the remaining fragment(s) of the segment. If it is part of an existing vertex, then it is split into two or more vertices. The required directed edges are added between the resulting vertices, maintaining the order of requests.

Suppose process X has generated request for consecutive blocks $\{A, B, C, D\}$ and followed by another request for $\{P, Q, R\}$. The segment graph at this stage (shown in Figure 3.2(a)) is consisting of two vertices labeled as u and v, and an edge from u to v. At this state, if process Y has generated a request for consecutive blocks $\{C, D, E, F\}$ then the vertex u

is split into two vertices *u'* and *u"*, and a new vertex *w* is added to the graph. Edges are added from *u'* to *u"* and from *u"* to *w*. The segment graph at this state is shown in Figure 3.2(b).

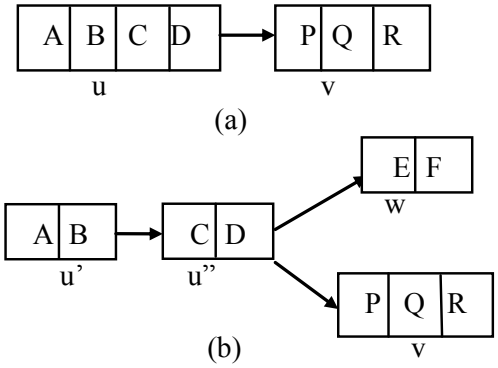


Figure 3.2: Graph based request segmentation - Graph after (a) Request ABCD and PQR arrived (b) Request CDEF arrived

In order to save memory, older vertices which are not referenced for a long time can be removed from the graph. When identifying the segments, linear sequence of vertices with contiguous LBA's and without any branches incident with the intermediate vertices are merged to get a larger segment. It may be noticed that in the constructed graph, each node is a possible segment of data blocks that can be mapped to a bucket with similar segments.

3.1.4 Similar segments identification

The segments, which are consecutive sequence of blocks are indexed using similarity based indexing. Metadata of all similar segments along with the fingerprints of the blocks, physical block mappings and their reference counts are stored in the buckets. Similarity of segments is identified using Broder's theorem [80].

Let $S1$ and $S2$ be two segments with $H(S1)$ and $H(S2)$ representing the sets of fingerprints of the data blocks of $S1$ and $S2$ respectively. H is chosen uniformly and at random from a min-wise independent family of permutations. Let $min(H(S1))$ and $min(H(S2))$ denotes minimum fingerprint of set $S1$ and $S2$ respectively. According to Broder's theorem, if two segments are highly similar, they share many blocks and hence the probability

that their minimum fingerprint is same is very high and is same as their Jaccard similarity coefficient as given in Equation 3.1.

$$Pr[min(H(S1) = min(H(S1)] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{3.1}$$

HDS identifies segments and computes fingerprints for all of the blocks of that segment, using MD-5 algorithm. RID of the segment is determined and the corresponding bucket is identified. If the bucket is not existing, a new bucket is created and metadata of the segment is added to that bucket. Otherwise, selective deduplication of the segment is performed, and the resulting metadata of the segment is added to the bucket. Similarity based segment indexing confines the search space to the respective bucket while identifying duplicate blocks of a segment and hence minimizes the metadata lookup overhead.

3.1.5 Metadata management

Before describing the deduplication algorithms, in this subsection metadata structures used in HDS are presented. The key data structures used in HDS are – Bucket which consists of metadata of similar segments, RID index table to index the buckets, LBA-to-PBA and PBA-to-Bucket mapping tables and Hash table. Buckets are organized as an array of structures with the members - segments of physical blocks, their fingerprints and reference counts. If a bucket is overflowed, due to the addition of more number of segments, link bucket is appended. By using chain of buckets, metadata of large number of similar segments can be stored together. RID index, LBA-to-PBA, and PBA-to-Bucket mapping tables are organized as B-trees. While processing read requests, LBA-to-PBA mapping table is used to locate the required target physical blocks.

Metadata of small segments, is stored in hash table to enable faster access and reducing the overhead while performing deduplication of small requests. Hash table entry consists of block number, fingerprint and reference count. Fields of the metadata structures are shown in Figure 3.3.

Buffer cache: All modern operating systems make use of disk buffer cache, for improving the performance of I/O system. In HDS, small amount of buffer cache is reserved

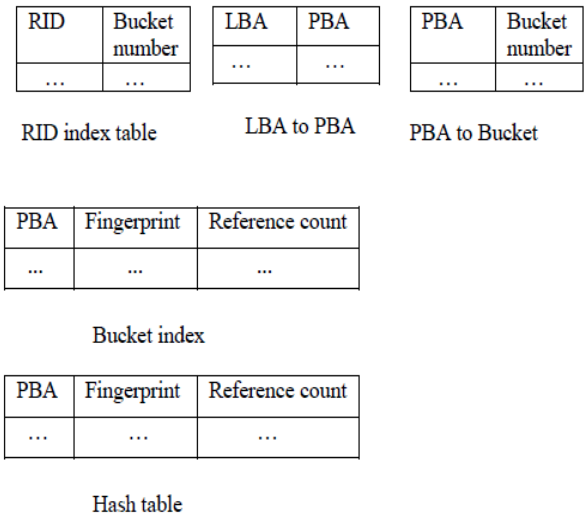


Figure 3.3: Metadata structures

for LBA buffers and remaining is used for PBA buffers. Read requests are processed by first searching in the LBA buffer cache. If data is not found in LBA buffer cache, then LBAs are mapped to the corresponding PBAs using cached or on-disk metadata structures. Using those mapped PBAs, PBA buffer cache is searched for the data. Write requests are processed quickly by copying the data into LBA buffer cache. LBA buffer cache may contain duplicate data, but PBA buffer cache contains mostly unique data, which makes the efficient utilization of the buffer cache. LBA buffer cache content is deduplicated in the background, which does not affect the performance of write requests. In order to improve the performance, metadata is also cached within main memory. Portion of buffer cache is reserved for metadata caching. At present, both of the data and metadata caches are using LRU cache replacement policy. On disk meta data structures are updated only when the cached copy is replaced or cache flush operation (once for every 30 seconds interval) is applied.

3.1.6 Selective deduplication

In primary storage systems, small requests normally dominate over large requests and most of them are duplicate [5][6]. The iDedup [8] deduplication system has ignored small request deduplication by stating that the capacity saving is insignificant and deduplication

overhead and latency per block is more for small requests. Identification of duplicate requests causes increase in latency for individual requests. But, it is observed in the present study that the overhead for the elimination of duplicate write requests is much lower than that of duplicate disk I/O requests. Hence, the deduplication of small I/O requests enhances overall I/O performance. For large write request deduplication, the segments of requests are identified and deduplication is applied on those segments. Different approaches are followed in HDS for deduplication of small and large segments of requests which are described in the following algorithms. Abbreviations used in the algorithms are given in the Table 3.1.

Table 3.1: Abbreviations used in the algorithms

Notation	**Description**
r	Request
d	Data block
D	Set of fixed size data blocks
f	Fingerprint of a block
s	Segment
S	Set of segments
F_s	Set of fingerprints of segment s
B	Bucket

3.1.6.1 Small write request deduplication

Initial step is to compute fingerprint for content of data block using MD-5 hashing algorithm. Hashing is applied on the fingerprints of the data blocks to locate the slots in the hash table. For small write requests, searching in the hash table for duplicate blocks and updation of metadata can be done faster. Thus, while deduplicating a small request, the latency for processing is minimized using hash table. Detailed steps for the deduplication of small write requests are shown in algorithm 3.2.

Algorithm 3.2 Small write request deduplication

Input: Request *r(LBA address, size, data)* from SW queue.

1: $f \leftarrow$MD-5(d)
2: Search hash table.
3: **if** f is found in the hash table **then**
4: $\quad reference\_count++$ if necessary.
5: **else**
6: $\quad$ Add new entry in hash table slot.
7: $\quad reference\_count \leftarrow 1$
8: $\quad$ Update metadata.
9: **end if**

3.1.6.2 Large write request deduplication

Large request deduplication improves storage capacity. However, full deduplication can lead to disk bottleneck and data fragmentation and consequently performance of the system gets affected. In order to reduce the data fragmentation, selective deduplication is applied in case of large request deduplication.

Initial step is to identify segments of a write request using graph-based request segmentation module. For each segment, fixed-size chunking is applied to generate fixed-size data blocks. For each block, fingerprint is computed using MD-5 hash algorithm. Minimum hash, among all the hashes of chunks, is taken as RID of the segment. The computed RID is searched in the RID index table to locate the bucket of similar segments and that bucket is searched for matching fingerprints of the segment. The mapping of duplicate blocks to the existing blocks in the bucket may result in a situation, where the segment being considered may be fragmented into smaller segments. If the number of blocks in such smaller segments is beyond a predefined threshold value (less than 3 blocks), the duplicates are not eliminated and the original segment with the duplicate blocks is added to the bucket. However, if the fragmentation is below the threshold, then duplicate blocks are mapped to the existing blocks in the bucket and only unique blocks are written. Metadata of the unique blocks is added to the bucket. Detailed steps for the deduplication of large write requests are shown in algorithm 3.3.

The use of different approaches for small and large requests with selective deduplication minimizes the overhead for small requests and improves the storage efficiency and I/O performance for large requests. Small and large read requests are processed similarly but the former is given higher priority than the later.

Algorithm 3.3 Large write request deduplication

Input: Request *r(LBA address, size, data)* from LW queue.

Request r is processed and Segment graph is updated by
    graph based request segmentation module.
for each $s \in S$ **do**
    Perform fixed-size chunking of data to get D.
    for each $d \in D$ **do**
        $f \leftarrow$MD-5(d).
        $F_s \leftarrow F_s \cup f$
    end for
    RID$\leftarrow Minimum(F_s)$
    if RID hits index table **then**
        Obtain respective bucket number i.
    else
        Allocate new bucket i for segment.
        Update RID index.
    end if
    Get bucket B_i.
    Search for F_s in B_i.
    Construct PBA segment(s).
    Find the lengths of PBA fragments.
    if fragment length $\leq$ fixed threshold **then**
        Store the metadata of the segment without
        duplicate block elimination.
    else
        Add unique block's information and update
        metadata.
    end if
    Write unique data blocks to storage and
     update metadata.
end for

3.1.6.3 Read request processing

For a given read request, LBAs are searched in LBA buffer cache. If all LBAs are found, data is constructed and sent. Otherwise, the LBAs are mapped to PBAs and PBA buffer cache is searched for the data. If found, data is constructed and sent. If data is not found in both LBA and PBA buffer caches, then disk read is issued to read data blocks into the buffer cache and the data is assembled and returned. Detailed steps for processing read requests are given in Algorithm 3.4.

Algorithm 3.4 Read request processing

Input: Request *r(LBA address, size)* from SR or LR queue

```
1:  Search in the LBA buffer cache.
2:  if found then
3:      Construct data and return.
4:  end if
5:  Map LBA's to PBA's.
6:  Search in the PBA buffer cache.
7:  if found then
8:      Construct data and return.
9:  else
10:     Generate disk read request to obtain data blocks.
11:     Assemble data blocks into buffer and
12:       return buffer data.
13: end if
```

3.2 Experimental results and evaluation

Prototype of the HDS, Full deduplication system and native (without deduplication) systems are implemented and simulated under the Linux operating system running on Intel i7 processor based system, with standard I/O traces taken from three production systems at FIU as input. The input includes the I/O requests generated by the virtual machines running web server (Web), file server (Home) and email server (Mail) [22], for a duration of 21 days. Table 3.2 shows counts of I/O requests, LBAs, duplicate blocks and unique blocks

for all three data sets. It is observed that Mail, Web and Home traces have 14.32%, 19.1% and 18.71% of duplicate blocks respectively and each data block is accessed repeatedly many times (LBAs vs Total requests).

Table 3.2: Trace statistics

	Mail	Web	Home
Total requests	460334027	14294158	17836701
Read requests	51348252	3116456	726464
Write requests	408985775	11177702	17110237
Total LBAs	14741706	549174	36340810
Duplicate data blocks	2110399	104870	9237083
Unique data blocks	12631307	444304	27103727
Working set size (KB)	58966824	2196696	145363240
% of duplicate data blocks (excluding multiple writes to the same block)	14.32	19.1	18.71

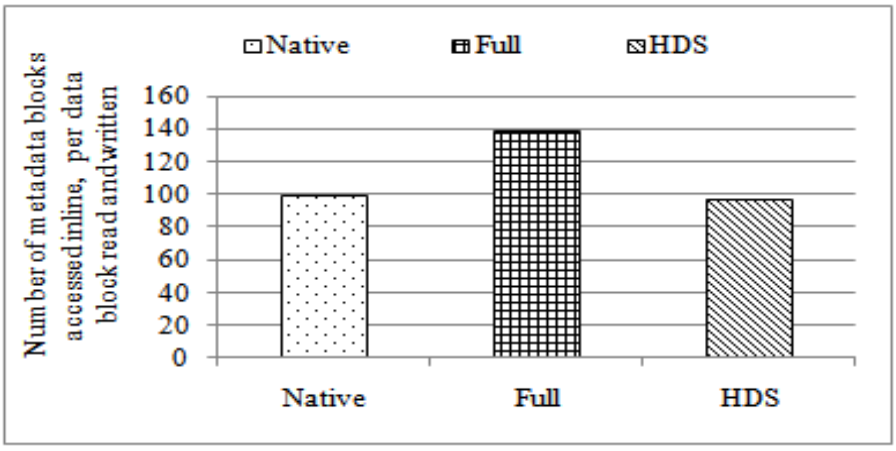
In the present experiments, 512 MB each for the data cache and metadata cache is used in the context of Home and Mail traces and 192 MB for data cache and 48 MB for metadata cache is used in the context of Web traces. Application of deduplication enhances overall storage efficiency and I/O performance. In the study, the parameters (i) normalized response time in terms of number of metadata blocks accessed inline, (ii) metadata overhead, in terms of the count of metadata operations (insert, delete, update and search operations), per data block, (iii) metadata overhead, in terms of number of metadata blocks accessed, per data block, (iv) write requests eliminated, (v) average overhead for read request, (vi) average data block segment length, (vii) normalized storage optimization and (viii) read and write response time statistics have been measured to assess the performance of the proposed HDS. Average data block segment length is used as a measure for fragmentation, with the assumption that longer the segment length, lesser the fragmentation. Response

time is taken as a measure for latency.

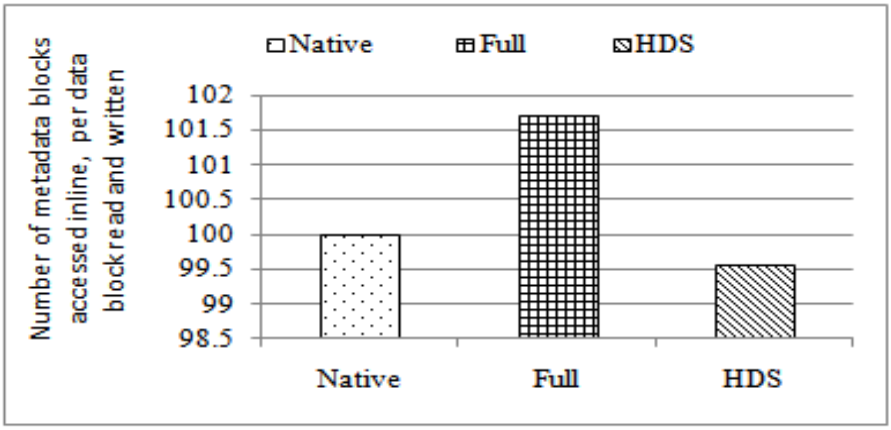
Performance of the HDS is compared with Full deduplication system. Normalized response time, which is important for a primary deduplication system, is compared with the state-of-the-art iDedup and POD primary deduplication systems. Other deduplication systems have reported estimated average response time in milliseconds for the read and write requests. We have considered a more general parameter, number of metadata blocks accessed inline, per data block read and written, which is a storage system independent parameter. Response times measured for native system, full deduplication system and HDS are shown in Figures 3.4a, 3.4b and 3.4c. Response time of Full deduplication system as compared to native system is degraded for all datasets. This is due to inline exhaustive search, performed by full deduplication system, which leads to increased metadata overhead. In HDS, deduplication is applied in the background. Due to this, the effect of overhead on inline request processing is not visible. Moreover, due to elimination of duplicates, normalized response times are observed to be improved compared to native system. Response time in case of Home dataset is very high for full deduplication system, due to repeated rewriting of the same small data blocks (512 bytes). Each rewrite causes the corresponding metadata to be updated several times. Whereas, in HDS, repeated rewrites result in LBA buffer modifications only. When delayed deduplication is applied in the background final copy of the data block is considered and metadata is updated only once. Thus unnecessary metadata updates, that may be generated due to intermediate data block modifications, are avoided. So compared to full deduplication system, HDS has shown much better response time even for Home traces.

Overhead of metadata accesses is measured through two parameters: number of metadata blocks read/written and count of metadata operations per data block read/written. Overheads measured in terms of metadata blocks read/written per data block read/written are shown in Figure 3.5a, 3.5b and 3.5c. for Mail, Web and Home datasets respectively. It can be observed that metadata overhead of HDS is much lower than that of full deduplication system. The overhead values are proving the applicability of the proposed system for inline deduplication.

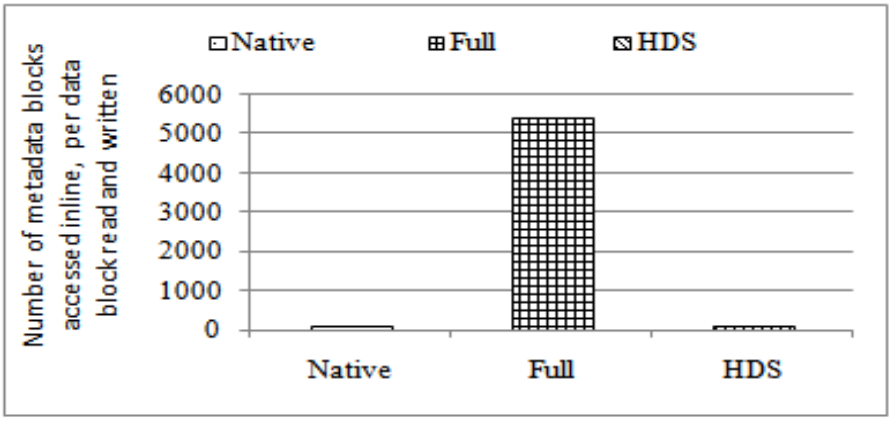
Overheads measured in terms of count of metadata operations is shown in Figures 3.6a,



(a) Mail dataset

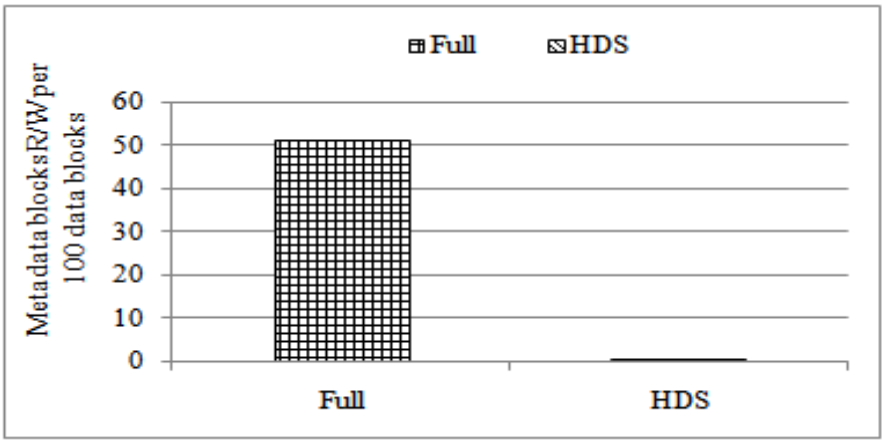


(b) Web dataset

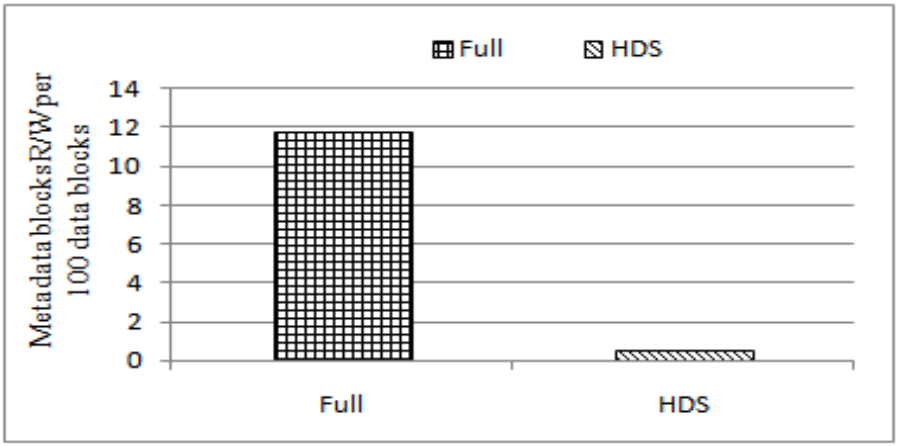


(c) Home dataset

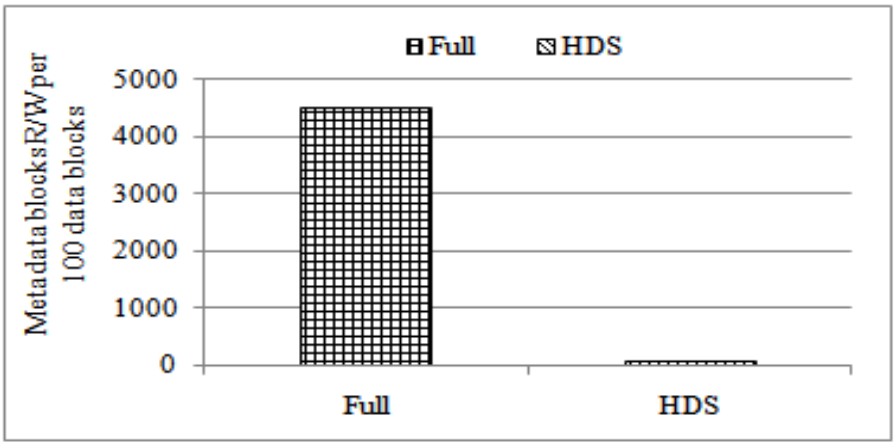
Figure 3.4: Normalized response time (in terms of number of metadata blocks accessed inline)



(a) Mail dataset

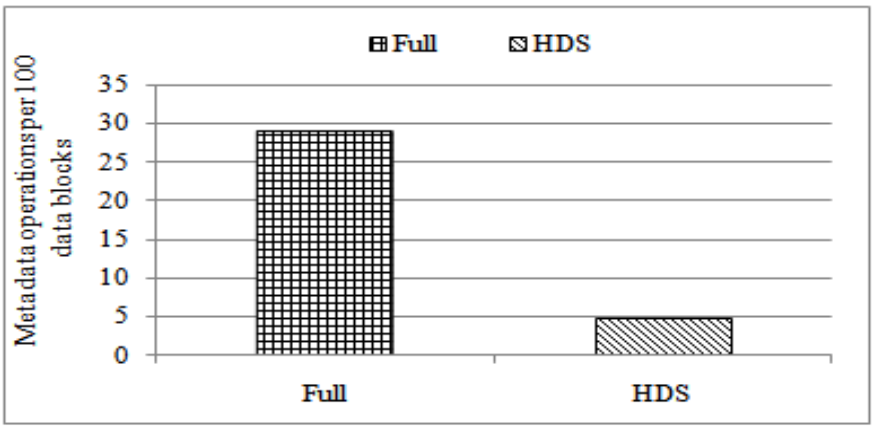


(b) Web dataset

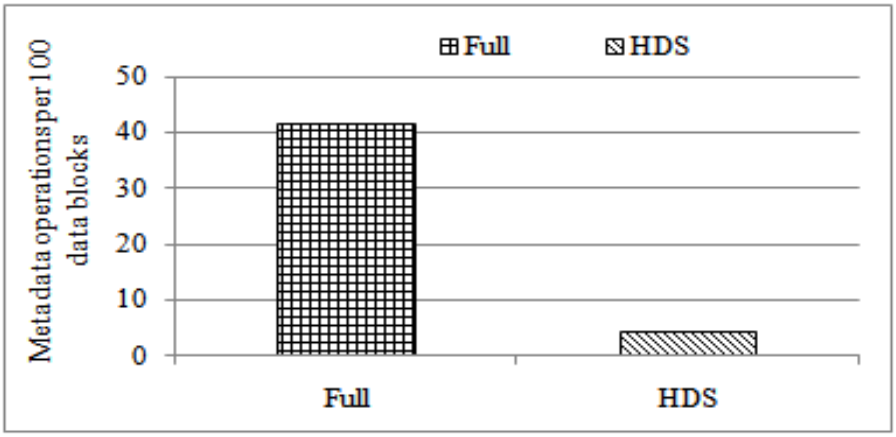


(c) Home dataset

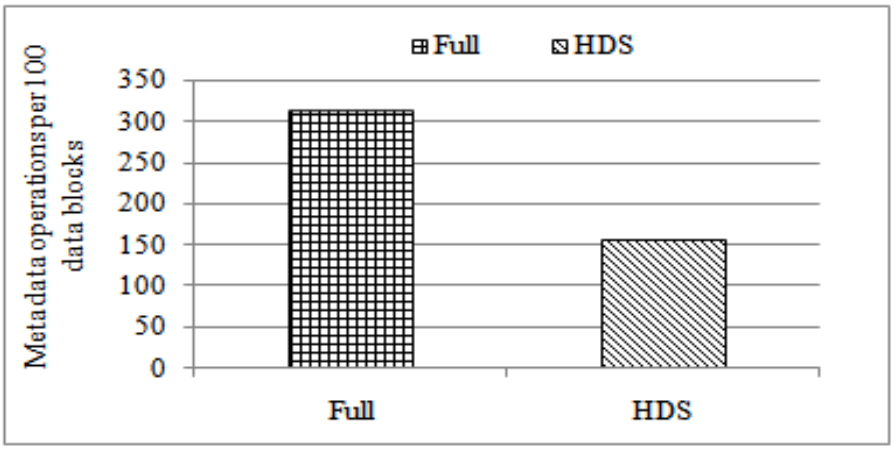
Figure 3.5: Metadata overhead (in terms of count of blocks)



(a) Mail dataset



(b) Web dataset



(c) Home dataset

Figure 3.6: Metadata overhead (in terms of count of operations)

3.6b and 3.6c. for Mail, Web and Home datasets respectively. It can be observed

that, with HDS, average metadata operations is reduced by factors of 83.24%, 89.21% and 49.86% for Mail, Web and Home traces respectively, compared to full deduplication system. As metadata of similar segments is stored together in buckets, the search space is confined to a bucket. Thus the metadata overhead is reduced in the proposed HDS compared to full deduplication system. For both of these overhead parameters relatively higher overhead can be observed in the case of Home traces. It is observed that, for Home traces same smaller blocks (Block size is 512 bytes), are repeatedly rewritten, that have generated new fingerprints. In order to check for the uniqueness of these fingerprints (block data), more metadata blocks need to be accessed.

Percentage of write requests eliminated by applying deduplication is shown in Figure 3.7. It is observed that 20.32%, 15.84% and 54.56% write requests for Mail, Web and Home datasets respectively are eliminated using full deduplication system. Whereas, HDS has eliminated 7.98%, 5.27% and 23.69% of write requests for Mail, Web and Home datasets respectively. The decrease in eliminated write requests in HDS can be attributed to the missed duplicate blocks in the process of selective deduplication. However, the multi-fold gain in the performance outweighs the reduction in duplicate request elimination.

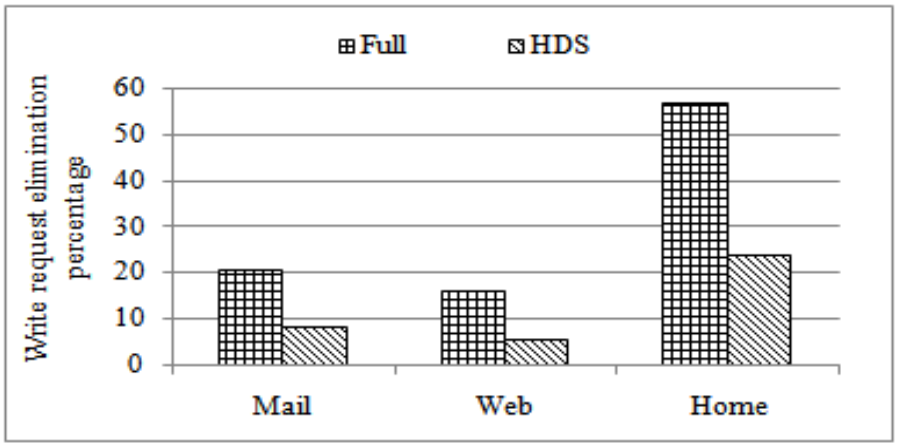


Figure 3.7: Write request elimination

In order to measure data fragmentation due to deduplication, average length of data block segments is computed. Figure 3.8, shows average segment length values for all three data sets. As selective deduplication along with reduction in fragmentation is applied, the proposed HDS has generated 1.7, 4, and 1.1 times longer segments for Mail, Web

and Home datasets respectively, compared to full deduplication system. The reduction in fragmentation improves the I/O system performance, particularly for sequential accesses.

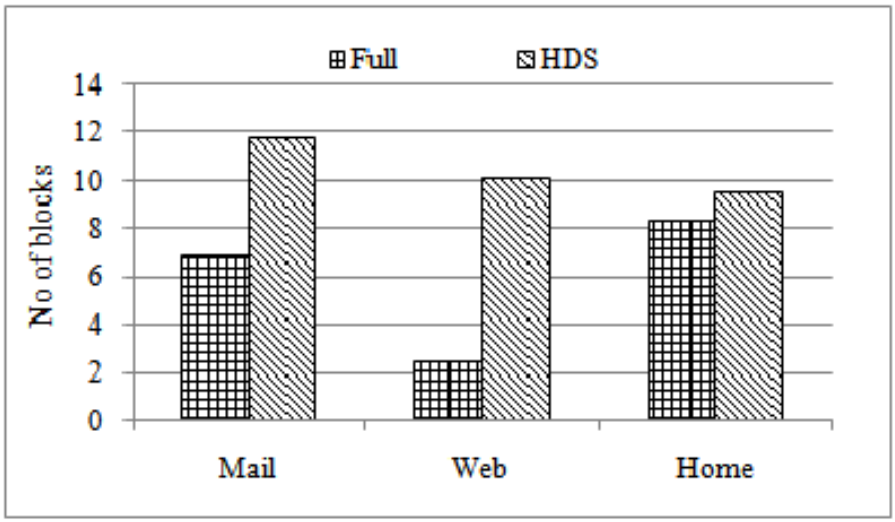


Figure 3.8: Average segment length

Read overhead per data block read operation is given in Table 3.3. It is negligible value for HDS. As increase in read latency due to deduplication is negligible, applications can run seamlessly. This shows that HDS is best suited as primary storage deduplication system.

Table 3.3: Read overhead per data block read

	Mail	**Web**	**Home**
HDS	0.0027	≈ 0	0.0135

Storage optimization is measured by counting the total unique block writes issued by the processes and actually stored unique blocks after deduplication. Let T indicates total unique writes issued and S denotes total number of unique blocks stored then

$$\text{Storage optimization} = \frac{(T - S)}{T} * 100 \tag{3.2}$$

Saving of storage space by applying deduplication is shown in Figure 3.9. It can be observed that space saving with full deduplication is 71.3%, 92.75% and 49.61% for Mail,

Web and Home datasets respectively. However, in case of HDS, the space saving is observed to be 69.3%, 91.6% and 44.65% for Mail, Web and Home datasets respectively. Reduction in space saving due to selective deduplication, is negligible, but the performance gain is more significant.

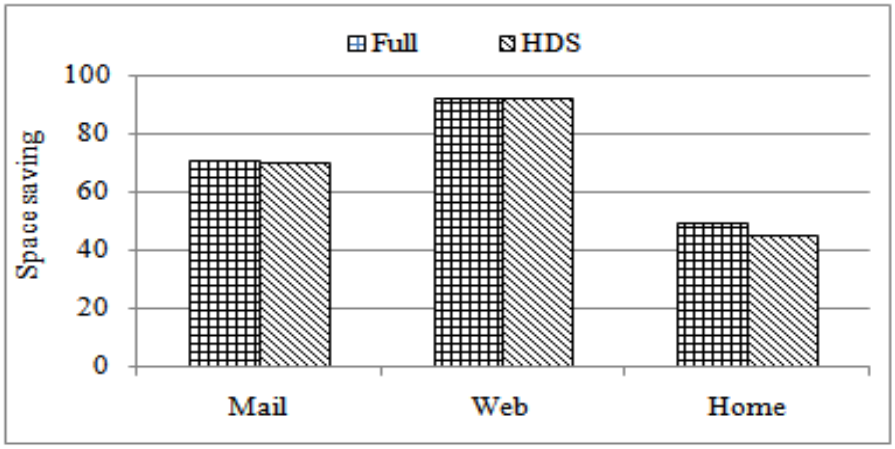
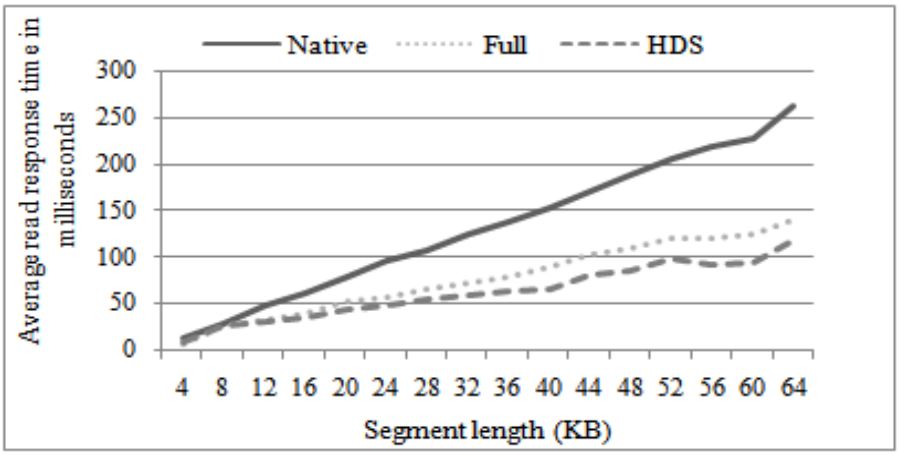
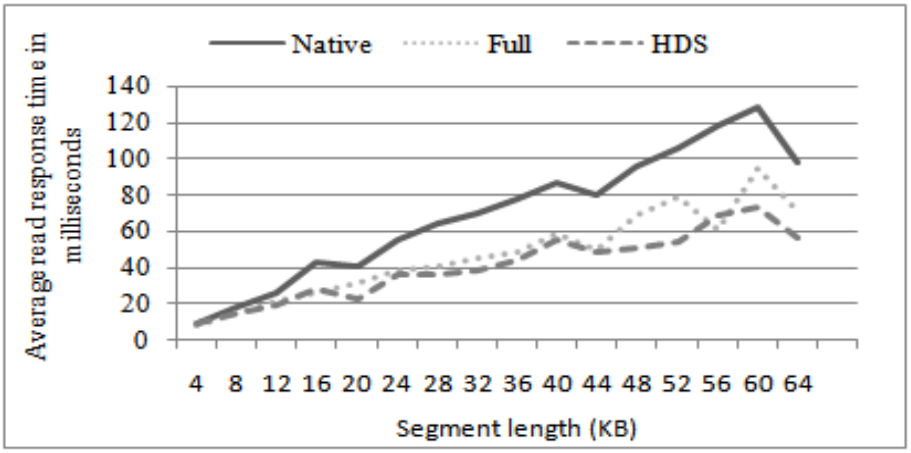


Figure 3.9: Storage space saving

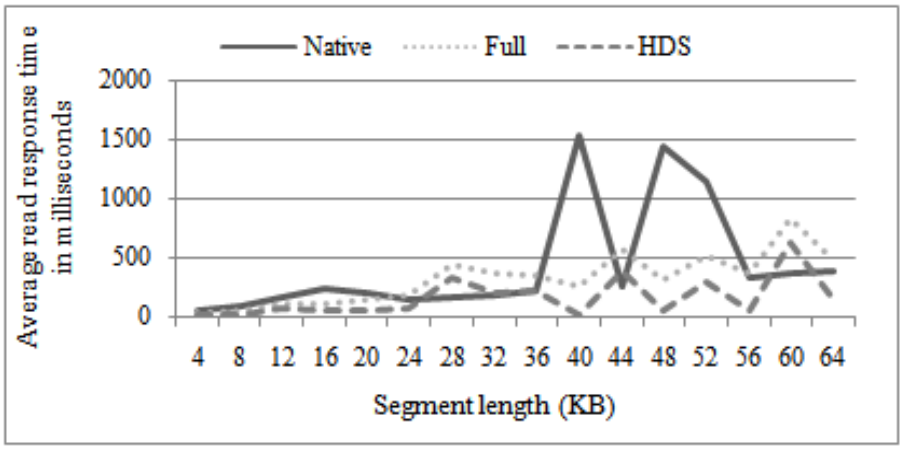
Average read response times, measured segment lengthwise, for native system, full deduplication system and HDS, are shown in Figure 3.10a, 3.10b and 3.10c. for Mail, Web and Home datasets respectively. In order to measure read response time, system call *clock_gettime() (CLOCK_REALTIME)* is used, which returns CPU time stamp value based, high precision time. Except the data block reading time, all other metadata processing time is measured in realtime, including actual metadata reading from disk (if a metadata cache miss occurs). It can be observed that average read response time of HDS is improved compared to native system due to better utilization of cache memory. Compared to full deduplication system also HDS shows slightly improved read response times, which can be attributed to the reduced data fragmentation in HDS. Standard deviation of read response times for native system, full deduplication system and HDS are shown in Figure 3.11a, 3.11b and 3.11c. respectively. It can be observed that the variation in read response time is within reasonable limits. Standard deviation of read response time for Home dataset shows zero variation for some request sizes. This is because of the presence of single requests for the corresponding request sizes. Total read requests are 2123 only and the remaining are write requests.



(a) Mail dataset

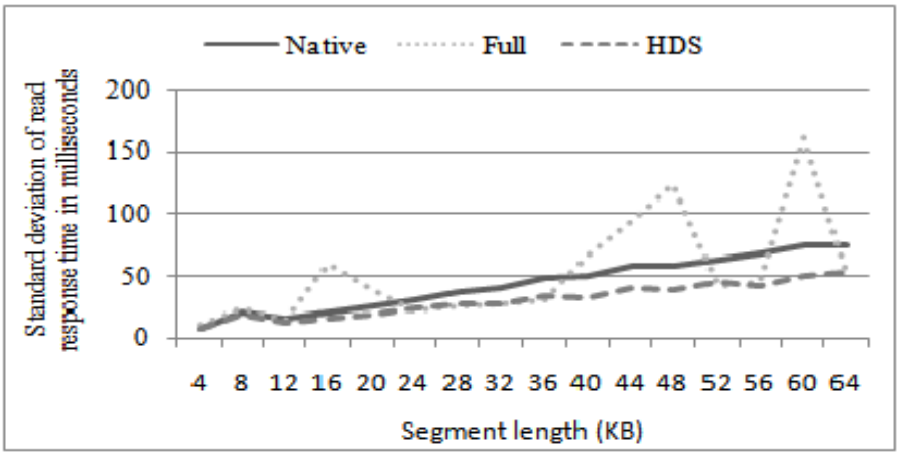


(b) Web dataset

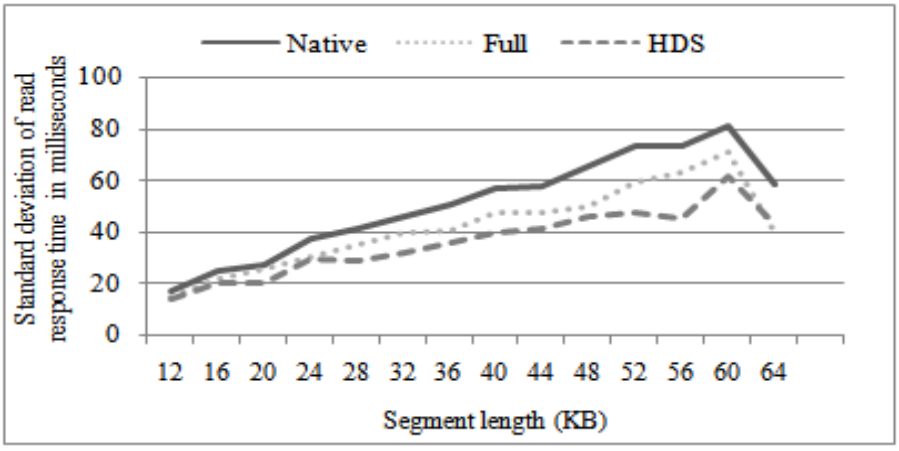


(c) Home dataset

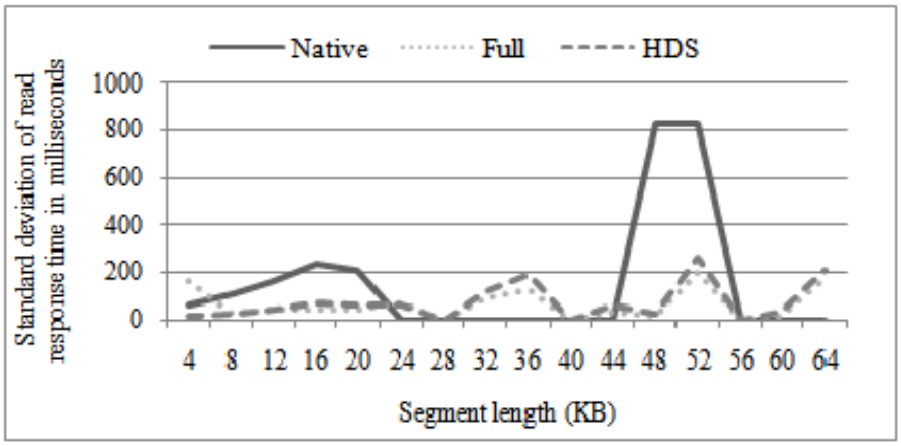
Figure 3.10: Average read response time (milliseconds)



(a) Mail dataset



(b) Web dataset



(c) Home dataset

Figure 3.11: Standard deviation of read response time (milliseconds)

Average write response times, measured for native system, full deduplication system and HDS, are given in Table 3.4, Table 3.5 and Table 3.6 for Mail, Web and Home datasets respectively. For the Home traces all writes issued are of the same size, i.e., 4 KB. So for the comparison of performance, only 4 KB size is considered. But in HDS, while applying deduplication, longer segments are used. For the Web dataset, HDS has used maximum of 16 KB data segments for deduplication in the background, in order to improve the deduplication ratio. While measuring the write response time, fingerprint computation time is assumed to take 32 μs, per 4 KB block [14]. All other processing time is actual metadata processing time, which is measured using *clock_gettime()* system call. Data block accessing time is measured based on the physical dimensions (Cylinders/Heads/Tracks) of the disk unit, on/off state of the disk motor, and current position of the read/write head. HDS performs data deduplication in the background. We have measured the time requirement of deduplication of data segments, segment lengthwise, separately from that of foreground write request processing. Foreground write request processing time is varying from few tens of microseconds to few hundreds of microseconds depending on the size of the request. The delay is due to buffer allotment and data copying, which is essential in any system. Background processing of deduplication, on different sizes of data segments, is in the order of few hundreds of microseconds. This time is excluding the actual data block writing time, that may happen at the time of dirty block flushing. It can be observed that the write response time of full deduplication system is almost close to or better than that of native system. In the full deduplication system, as the metadata does not exhibit locality property, frequent disk accesses are required which causes increase in write response time. Whereas, in HDS caching is based on similarity, which exhibits better hit ratio and hence write response time is in the order of few hundreds of microseconds. It may be noted that the experiments are conducted on an idle machine and dirty data flushing time is not included while measuring the response times for any of the reported systems. Overall whenever native system write request requires cache replacement and the victim block is a dirty one, then disk access is required. Similarly for full deduplication system even though the data cache utilization is better than the native system, metadata accesses may require additional disk accesses. Whereas, for HDS, due to similarity based grouping, such

metadata disk accesses are very minimal.

Table 3.4: Mail dataset write response time

Segment Length (KB)	Native (ms)	Full (ms)	HDS Inline write (μs)	HDS Background Dedup (ms)
4	3.41	2.63	40.92	0.28
8	4.69	3.81	54.86	0.50
12	9.01	6.62	102.24	0.46
16	13.37	11.41	155.68	0.47
20	18.01	14.43	212.41	0.48
24	20.87	17.65	248.31	0.52
28	24.58	20.20	292.07	0.48
32	28.39	24.02	338.94	0.58
36	33.23	30.75	411.05	0.62
40	37.41	35.56	451.93	0.73
44	38.25	32.48	457.25	0.64
48	41.57	33.13	500.68	0.70
52	45.26	39.03	545.63	0.82
56	48.83	42.92	591.21	0.90
60	56.47	48.30	701.71	0.98
64	53.53	40.76	651.60	1.24

Table 3.5: Web dataset write response time

Segment Length (KB)	Native (ms)	Full (ms)	HDS Inline write (μs)	HDS Background Dedup (ms)
4	1.05	0.96	12.54	0.0001
8	1.15	1.08	18.47	0.0002
12	9.26	9.08	101.41	0.0009
16	18.20	18.40	185.03	0.0005
20	14.55	15.04	175.58	
24	31.59	24.83	330.05	
28	32.73	29.06	360.57	
32	31.40	27.41	354.08	
36	38.95	36.53	445.28	
40	31.40	35.36	391.66	
44	26.52	26.17	311.41	
48	17.67	20.12	251.57	
52	26.48	32.59	378.17	
56	23.92	27.38	303.98	
60	21.67	28.35	297.34	
64	42.42	57.18	621.05	

Table 3.6: Home dataset write response time

Segment Length (KB)	Native (ms)	Full (ms)	HDS Inline write (μs)	HDS Background Dedup (ms)
4	4.59	17.20	10.53	0.44

Standard deviation of write response times for native system, full deduplication sys-

tem and HDS are shown in Table 3.7, Table 3.8 and Table 3.9 for Mail, Web and Home datasets respectively. It can be observed that the variation in write response times is within reasonable limits.

Table 3.7: Mail dataset standard deviation of write response time

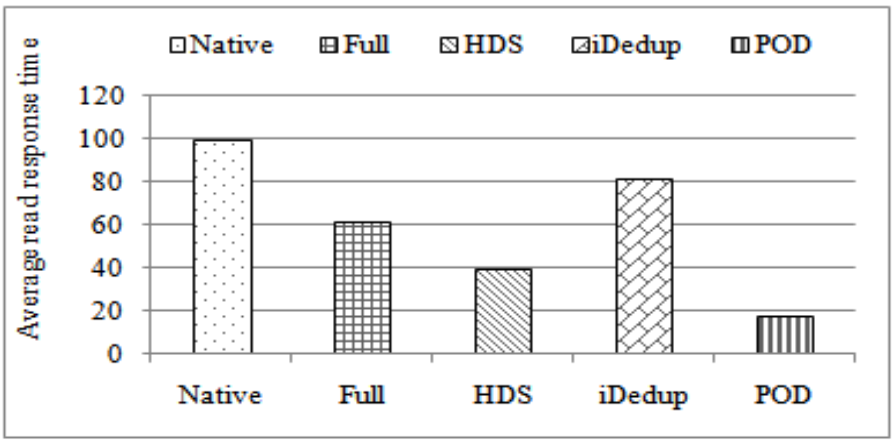
Segment Length (KB)	Native (ms)	Full (ms)	HDS Inline write (μs)	HDS Background Dedup (ms)
4	2.38	9.37	34.64	1.49
8	3.49	17.36	49.14	4.01
12	7.28	25.30	96.59	2.75
16	9.36	43.91	124.91	1.95
20	10.12	27.03	144.95	2.45
24	11.71	51.39	170.17	1.95
28	13.54	40.40	197.77	1.81
32	15.12	49.83	220.82	1.75
36	14.86	68.76	230.20	1.88
40	17.26	82.71	256.99	1.93
44	20.21	64.44	294.38	1.64
48	22.52	51.88	322.75	1.63
52	23.62	70.45	345.76	1.76
56	25.10	61.16	365.99	1.80
60	22.75	89.30	356.53	1.81
64	29.78	44.76	422.10	2.14

Table 3.8: Web dataset standard deviation of write response time

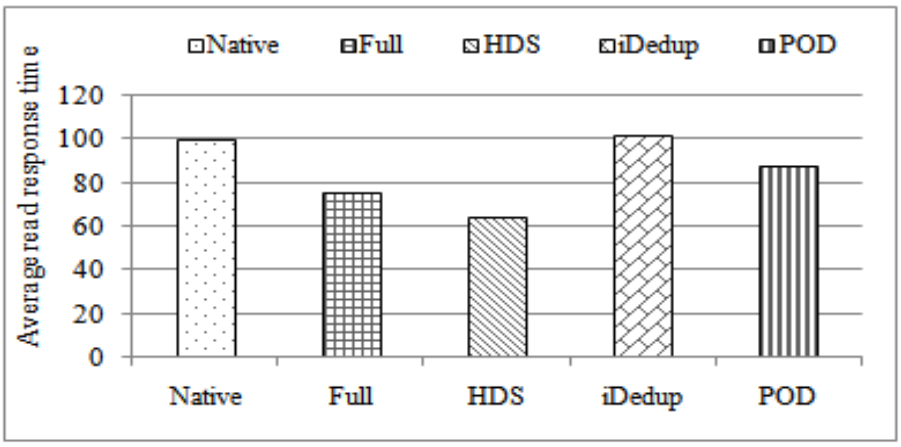
Segment Length (KB)	Native (ms)	Full (ms)	HDS Inline write (μs)	HDS Background Dedup (ms)
4	6.47	6.12	67.98	0.01
8	7.67	7.21	82.50	0.01
12	19.12	20.03	203.27	0.03
16	27.54	29.17	282.50	0.02
20	30.18	31.33	312.60	
24	33.93	31.66	381.18	
28	40.42	38.38	444.63	
32	37.35	35.17	439.03	
36	43.84	44.12	482.77	
40	44.42	45.09	498.47	
44	40.52	42.17	490.88	
48	38.72	43.31	464.73	
52	35.17	46.53	486.17	
56	51.17	46.64	531.29	
60	44.28	52.31	534.27	
64	38.95	57.13	559.49	

Table 3.9: Home dataset standard deviation of write response time

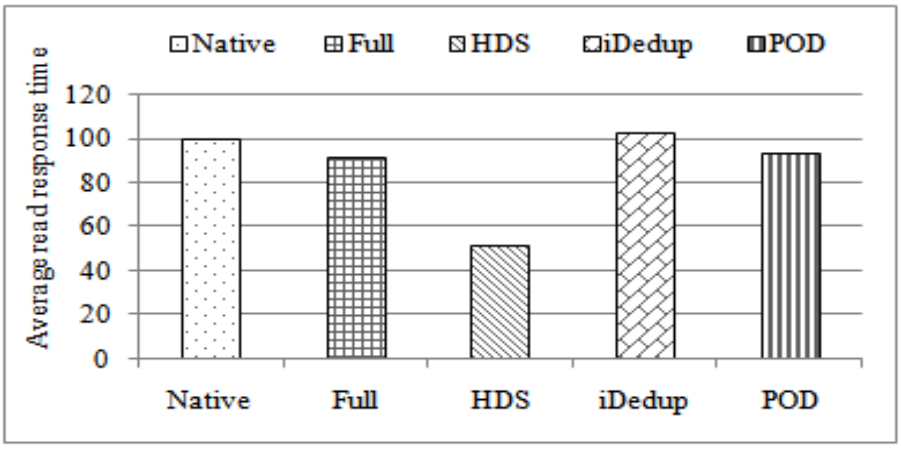
Segment Length (KB)	Native (ms)	Full (ms)	HDS Inline write (μs)	HDS Background Dedup (ms)
4	11.16	275.23	71.83	1.72



(a) Mail dataset



(b) Web dataset



(c) Home dataset

Figure 3.12: Normalized average read response time

Overall normalized average read response times of Mail, Web and Home dataset are

compared with that of iDedup and POD deduplication systems, and the corresponding observations are shown in Figure 3.12a, 3.12b and 3.12c. We have observed good improvement compared to native system due to better utilization of cache after deduplication. At the same time, HDS tries to reduce the data fragmentation also. Because of that, most of the times HDS read response time is better than other state-of-the-art deduplication systems. Write response times are not comparable, but the observed values shows that HDS is best suited as a primary deduplication system. HDS least affects, the read and write response times of the user applications.

3.3 Summary

HDS is a block based hybrid primary storage deduplication system which applies deduplication in the background. Main drawbacks associated with primary storage deduplication systems are disk-bottleneck and data fragmentation. Disk-bottleneck problem arises due to random access pattern of metadata. HDS uses similarity based indexing mechanism to locate all metadata of similar segments in one bucket and reduces the disk-bottleneck problem. Additionally, this type of indexing reduces search space for the identification of duplicates and because of the reduction in the overhead, HDS can be used for primary storage deduplication. HDS preserves the locality order of fingerprints in a bucket and the selective deduplication helps in reducing the data fragmentation. In turn, overall I/O system performance can be improved. At present, the HDS is designed to be used as a deduplication system for a single storage node. In future, this can be extended to support multiple storage nodes or distributed storage systems.

Chapter 4

Hybrid Deduplication System with Content-Based Cache for Cloud

The performance of centralized primary storage deduplication systems can be enhanced through various approaches such as indexing optimizations, caching optimizations, selective deduplication etc. In the previous chapter, we have proposed similarity-based indexing and selective deduplication to improve system performance. The performance of primary storage deduplication system at block level can be further enhanced through caching mechanism. Most of the existing works assumed strong temporal locality in workloads and used address-based cache with traditional cache replacement policies. There has been rare work in the context of caching optimization techniques for primary storage deduplication systems at the block level. Workloads on primary storage systems exhibit random access patterns and weak temporal locality. However, in the cloud environment, consolidation of different workloads causes interference of workloads which affects their cached locality. In order to utilize cache space efficiently, a content-based cache is required. To assess cache block popularity, cache replacement strategy prioritizes dynamically derived weighted reference counts and cache block staying period. This work proposes a buffer cache with a new cache replacement policy to improve system performance even more through caching mechanisms.

Primary storage systems are latency sensitive and have 20% to 70% data redundancy. Hence, applying resource-intensive deduplication tasks on I/O path incurs extra latency.

Thus, primary storage system deduplication found to be inappropriate previously [3][4]. However, recent works have shown that proper caching mechanisms can assist in improving system performance [11][14]. Primary storage deduplication systems apply deduplication on the write request path. As deduplication incurs additional processing, it leads to increased write request latency. Deduplication needs to access the metadata frequently to identify duplicates. Due to memory constraints, the metadata is stored on disk which results in frequent disk accesses. This is known as disk-bottleneck problem [16]. Various solutions are proposed in the literature, to reduce the metadata access overhead, such as similarity based indexing [3][7], locality based fingerprint caching [8], heuristic based approaches to group the fingerprints which are accessed together [9][10], usage of SSDs (Solid State Drives) for faster access of metadata [12], estimation of temporal locality of fingerprints [11], and workload specific cache sizing [14] etc. Existing works [8][11][14] have assumed that primary workloads exhibit strong temporal locality in data access and used address-based data cache. Hence, irrespective of the approach, they used temporal locality-based LRU policy for caching. However, address-based data cache may have duplicate content with different addresses. In the context of the cloud, the consolidation of different data-intensive workloads on to a small number of physical machines pose a new challenge for caching, which makes LRU replacement not very much suitable. Caching mechanism for cloud storage has to mainly address issues like duplicate data, interferences and mismatch of traditional cache replacement which are explained as follow.

- In cloud, different workloads are consolidated. These workloads may issue multiple I/O requests, having duplicate data block content with different addresses. As the cache is populated based on addresses of data blocks and oblivious to content, it may result in fetching and caching of duplicate content from the disk. As a consequence, the buffer cache cannot be utilized efficiently.

- The localities belonging to multiple workloads often interfere with each other. This interference affects the cached locality of individual workloads and hence, the workloads with weak temporal locality may cause cache replacement upon each miss, which may overwrite the current locality of other workloads.

- Deduplication of data causes a single data block to be shared among multiple workloads. Such data blocks exhibit different behaviour because of different access patterns of multiple workloads. As the shared data block is accessed by different workloads, access frequency changes over a period of time. Thus recency alone (Least Recently Used (LRU)) or frequency alone (Most Frequently Used (MFU) or Least Frequently Used (LFU)) based cache replacement policies are not effective in the cloud environment.

LRU is mainly designed for workloads that exhibit temporal locality. When the cache is full, LRU replaces the least recently accessed data block. In the cloud, consolidation of multiple workloads on primary storage systems exhibits random access pattern [3][2]. In this scenario, the cache may get populated with data blocks having a recent single reference, which leads to cache wipe out problem. In the context of frequency-based MFU replacement algorithm, the cache is populated with the data blocks that have low reference count, expecting that such blocks may be referenced again in near future. But if the prediction is wrong, then the cache remains occupied with unnecessary data blocks. In another frequency based replacement algorithm, LFU, the cache is populated with the data blocks that have high reference counts. In this approach the blocks which are heavily referenced in the past, but may not be referenced in the future, continue to occupy the cache. ARC [92] considers both recency as well as frequency. It uses LRU based two separate lists $T1$ and $T2$ whose size is dynamically adjusted based on the reference pattern. Recently accessed data blocks with single reference counts are placed in $T1$ list. If any block of $T1$ is accessed again, it will be placed in the $T2$ list. Thus, ARC avoids the wipe out of the cache, by keeping the single time referenced blocks in a separate list $T1$. But it does not consider the popularity of data blocks that may be referenced frequently at some intervals. The proposed Modified-ARC also contains two lists similar to ARC. But the entries in $T2$ are approximately ordered based on popularity value. Popularity parameter of a data block is computed using frequency of references and the last reference interval. The detailed algorithm is presented in the next section.

In this work, Hybrid Deduplication System (HDS) with similarity-based indexing and selective deduplication is proposed. To cope up with random access patterns and weak

temporal locality, a content-based data cache is proposed. In order to select the victim for cache replacement, popularity of the content, which is a combination of recency and frequency of references is considered. Every data block in the cache has associated metadata which contains information such as counts of read and write references and last access time etc. Metadata is maintained not only for the data blocks present in the cache but also for some of the recently evicted data blocks. Deduplication identifies and eliminates duplicate data blocks and enforces data cache to maintain unique blocks. The main contributions of this work are given below.

- Proposes similarity based indexing and selective deduplication.
- Maintains content-based data cache.
- Uses deduplication to avoid caching of duplicate data blocks.
- Introduces data block popularity metric based on weighted frequency, idle staying period and recency
- Proposes popularity metric based cache replacement policy to cope up with the weak temporal locality

The rest of the chapter is organized as follows. Motivation and background is presented in Section 4.1 and Section 4.2 gives a detailed explanation of the design of HDS with content based cache. Experimental results and evaluation are presented in Section 4.3. Finally, Section 4.4 concludes the work.

4.1 Motivation and background

In the cloud, workloads of different clients are consolidated on a few physical machines. Workloads may have I/O requests of different sizes and some of the requests may be duplicates. In order to improve I/O system performance, the cache is used. Generally, the cache is populated with recently or frequently accessed data which is usually determined based on data block addresses irrespective of content. There is a possibility of having many data blocks with the same content. When an I/O request arrives for such a data block, even

though the content may be available in the cache, but with a different address, leads to a cache miss. Existing cache replacement algorithms [92][93][94][95] are oblivious to content of the cache block. Hence may lead to unnecessary disk accesses. Mere recency or frequency-based cache replacement is also not useful, as access popularity of cache block changes over a period of time. There is a possibility of having data blocks whose access frequency is equal but their staying period in the cache and the last reference intervals may vary. Count of references, idle staying period and last reference interval (recency) are important factors in determining the popularity of a data block. In order to increase the cache utilization efficiency, deduplication can be used. Deduplication helps to maintain the unique cache content. Whenever there is a need for cache replacement, it considers a better parameter - popularity of the data block, rather its recency or frequency of reference.

The study of the standard FIU I/O traces has also motivated for the present work. FIU I/O traces [22] have the data blocks of size 4 KB for Mail and Web traces, and 512 Bytes for Home traces. Each trace consists of time-stamp in nanoseconds, process id, process name, LBA (Logical Block Address), count of blocks, write or read operation, major device number, minor device number and MD-5 hash value of the content. These traces are used by many researchers for studying about duplicate content and duplicate I/O requests. Overwriting the same data block (LBA) is considered as duplicate I/O request, and writing the same content to different LBAs is considered as duplicate content. Duplicate I/O requests and duplicate content statistics for the FIU traces are shown in Figure 4.1.

Based on these statistics, we can claim that the cloud has domination of duplicate data and duplicate I/O requests. Content-based cache allows more unique data to be cached compared to address based cache. To handle different workload's locality interferences, instead of LRU replacement policy, popularity metric based cache replacement algorithm can be used, which can improve the content-based cache performance further.

In literature, as shown in Table 4.1, deduplication systems in the cloud environment uses DRAM or SSD as a cache device. These cache devices can be used as address-based cache or content-based cache. ChunkStash [12], PDFS [3] and Stream Locality Aware DEduplication (SLADE) [11] systems used SSD as address-based cache for storing

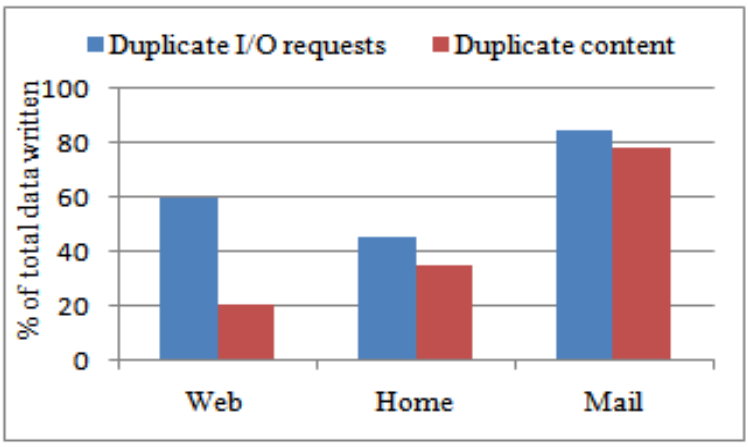


Figure 4.1: Duplicate content and I/O request statistics of FIU I/O Traces

Table 4.1: Existing deduplication systems classification based on cache type and cache device

Cache Device \ Cache Type	Address Based Cache (LRU Replacement)	Content Based Cache (Novel Replacement)
SSD	PDFS [3] ChunkStash [12] SLADE [11]	D-ARC [13] CacheDedup [96] FARC [97]
DRAM	iDedup [8] POD [14] HANDS[9]	This work

deduplication metadata. Flash-aware Adaptive Replacement Cache (FARC) [97], PLC-Cache [98] and Nitro [99] deduplication systems used SSD for data cache and addressed mainly SSD endurance problem. As SSD can undergo a limited number of write operations, the deduplication system can avoid duplicate data writing on SSD devices. Very few authors have worked on cache replacement policies for deduplication systems. Among them, CacheDedup [96] and D-ARC [13] deduplication systems have used novel cache replacement policies (variants of DARC), for solving the endurance problem of SSD.

Most of the existing deduplication systems that use DRAM as cache are based on workload locality assumption. iDedup [8], POD [14] and HANDS [9] have used DRAM as an

address-based cache with LRU replacement policy. Wildani et al. [9] and Wu et al. [11] have used heuristics to optimize caching. Koller et al. [30] and Sudevalayam et al. [100] have used DRAM as content-based cache with LRU replacement policy.

There are other cache replacement policies such as ARC [92], Two Queue (2Q) [93], Multi-Queue (MQ) [94] and LRU-K [95] . Some of them are frequency-based, recency based or both. 2Q maintains one FIFO queue and two LRU lists. First time accessed blocks are placed in the FIFO queue whereas, evicted and re-accessed blocks are maintained in LRU lists. MQ has multiple queues and uses access frequencies to determine the queue for block placement. LRU-K identifies the resident time of the cache blocks as the times of the K^{th}-to-last references to blocks. ARC uses recency and frequency properties to adapt itself to changing access patterns of workloads. But it ignores the staying period of the block in the cache. In the cloud, workload locality assumption vanishes due to consolidation of workloads and the traditional replacement policies are not suitable. Deduplication system for cloud storage using DRAM as content-based cache with novel cache replacement, under weak temporal locality, is not studied much in the literature. While computing the popularity (importance) of a block, in order to handle highly accessed blocks in the past, reference count along with their staying period and recency of references also should be considered. In the proposed Modified-ARC, all these three parameters are considered for computing the popularity of a cache block.

4.2 Design of hybrid deduplication system with content based cache

In Cloud, each physical machine runs several virtual machines, providing heterogeneous services such as web services, email etc. Consolidation of these different workloads generates I/O requests, which go through the file system to the block layer. Data relevant to each workload get accumulated at the block level. In order to utilize the buffer cache at the block level efficiently, deduplication can be applied at the block level.

Hybrid Deduplication System is a block-level primary storage deduplication system. As

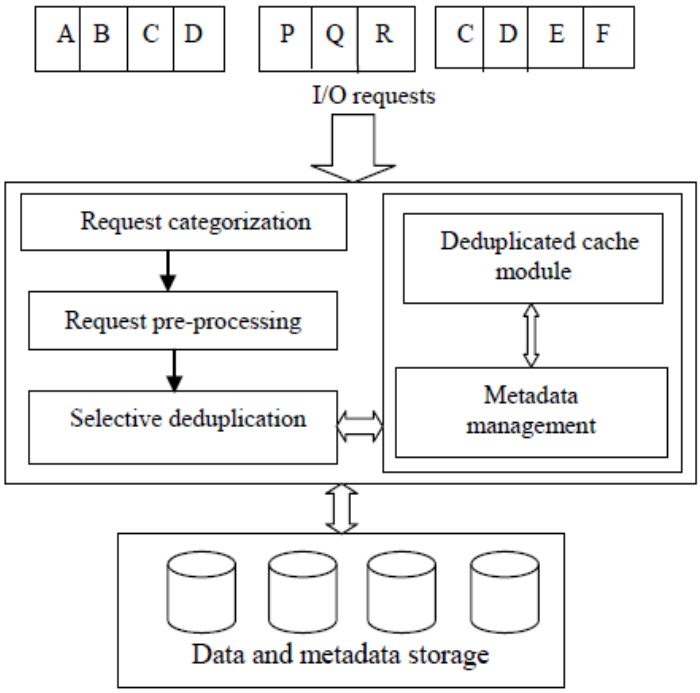


Figure 4.2: Hybrid deduplication system

shown in Figure 4.2, it consists of five modules such as request categorization, request pre-processing, selective deduplication, deduplicated cache module and metadata management.

Request categorization: This module is responsible for determining whether an incoming write request is a small-size or large-size request. If the size of the request is less than or equal to 8 KB, it will be treated as a small-size request otherwise it is treated as a large-size request.

Request pre-processing: Small-sized request undergoes fixed-size chunking of size 4 KB. Then the hash value is computed using the MD-5 hash algorithm. Large-size requests are segmented as 64 KB size segments. Each segment undergoes fixed-size chunking of size 4 KB. Thus each segment has 16 consecutive blocks. Hash value is computed for each block of the segment. Minimum hash among them is taken as the representative of the segment.

Metadata structures: Deduplication metadata maintains information such as the hash value of the block, its physical block address, logical block address and a reference count. Metadata is maintained for small-size requests and large-size requests separately. Small

request's metadata is stored in a hash table. Whereas, metadata for large requests is stored in the buckets. Bucket stores metadata of different segments with the same representative hash value.

Selective deduplication: Large write request deduplication is performed selectively. Each segment of a large request to be deduplicated, has a representative value that is used to locate the respective bucket. The segment is deduplicated, if the deduplication does not result in fragments of segments that are less than three blocks. Otherwise, the segment is written to the bucket without deduplication.

Deduplicated cache module: As deduplication is applied before storing the data in the buffer cache, this component is named as 'Deduplicated cache' module. After performing necessary deduplication operations, the data in the buffer cache, along with the metadata is written to the storage devices. It maintains mostly unique data blocks. For each of the blocks, information like write reference count, read reference count, last access time etc., are maintained. The first four modules are explained in more detail in our previous Chapter 3 and in this Chapter design and implementation of deduplicated cache module is given.

4.2.1 Deduplicated cache module

Deduplicated cache is a cache management module which is working at the block layer. It intercepts I/O requests from the selective deduplication module and performs deduplication to eliminate the duplicate data blocks. It consists of fixed-size data and deduplication metadata caches. Data cache is a content-based cache that maintains unique data blocks. Deduplication metadata cache is used to maintain indexing information that is useful for searching and identifying duplicates, and for mapping the Logical Block Addresses (LBA) to the fingerprints and the fingerprints to Data Block Addresses (DBA). Each cached data block maintains information such as its fingerprint, count of read and write references and last access time. Deduplicated cache module is shown in Figure 4.3.

When a read request is arrived, the LBA-to-FP mapping table is searched for each LBA and if entries are not found those are fetched from the on-disk mapping table. Using the fingerprint values of these entries, the FP-to-DBA mapping table is searched and if entries

are not found, those are also fetched from the on-disk mapping table. The corresponding data blocks are read from the disk, if not found in the data cache. Data is copied from the cached data blocks into the application buffer. Steps for read request processing are given in Algorithm 4.1.

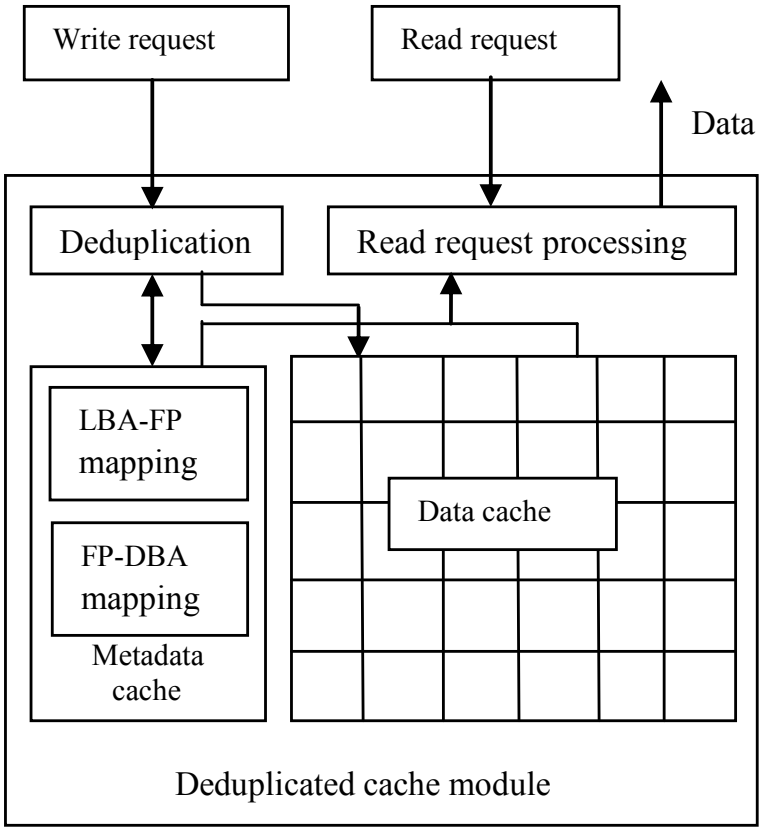


Figure 4.3: Deduplicated cache module

When a write request arrives, the data is stored temporarily in a buffer. Fingerprints are computed for each data block in the buffer, using the MD-5 hash algorithm. The LBA-to-fingerprint mapping table is searched for each block and fingerprints are updated if necessary. Reference counts of the data blocks of older fingerprints that are found in the fingerprint-to-DBA mapping table, are decremented. Similarly, reference counts of new fingerprints are incremented. During this process, the data blocks, for which reference counts reach zero, are deleted from the data cache. If a fingerprint is not found in the fingerprint-to-DBA mapping table, then a new entry is created. Steps for write request processing are given in Algorithm 4.2.

Algorithm 4.1 Read request processing

Input: Request *r(LBA address, size)*

1: **for** each LBA of the request **do**
2: Get $lbaentry$ of LBA.
3: **if** $lbaentry.fp \in FP-to-DBA$ **then**
4: $fpentry \leftarrow FP-to-DBA\ entry(lbaentry.fp)$
5: $fpentry.readref++$
6: **else**
7: Create new $fpentry$, if necessary by replacing the existing entry.
8: Read the corresponding data block from the disk.
9: $fpentry.readref \leftarrow 1$
10: **end if**
11: $fpentry.LAT \leftarrow current\ time$
12: Copy data from the block $fpentry.dbaddr$ to buffer.
13: Continue for the next block.
14: **end for**

Algorithm 4.2 Write request processing

Input: Request *r(LBA address, size, data)*

1: **for** each LBA of the request **do**
2: $fingerprint \leftarrow$MD-5(buffer)
3: **if** $LBA \in$LBA-to-FP **then**
4: $lbaentry \leftarrow LBA-to-FP\ Entry$
5: $fpentry\_old \leftarrow FP-to-DBAentry(lbaentry.fp)$
6: **else**
7: $lbaentry \leftarrow null$
8: $fpentry\_old \leftarrow null$
9: **end if**
10: **if** $fingerprint \in$FP-to-DBA **then**
11: $fpentry\_new \leftarrow FP-to-DBA\ entry$
12: **else**
13: $fpentry\_new \leftarrow null$
14: **end if**

Algorithm 4.2 continuation

15: **if** $fpentry\_old \neq null$ **then**
16: $fpentry\_old.count - -$
17: **if** $fpentry\_old.count == 0$ **then**
18: Delete the $fpentry\_old$ and data block.
19: **end if**
20: **end if**
21: **if** $lbaentry \neq null$ **then**
22: $lbaentry.writeref + +$
23: $lbaentry.LAT \leftarrow current\ time$
24: **else**
25: Create new *lbaentry*
26: $lbaentry.readref \leftarrow 0$
27: $lbaentry.writeref \leftarrow 1$
28: $lbaentry.LAT \leftarrow current\ time$
29: **end if**
30: **if** $fpentry\_new \neq null$ **then**
31: $fpentry\_new.writeref + +$
32: $fpentry\_new.LAT \leftarrow current\ time$
33: **else**
34: Create a new entry in $FP - to - DBA$ ($fpentry\_new$)
35: Allocate a new data block, if necessary replace existing block.
36: $fpentry\_new.readref \leftarrow 0$
37: $fpentry\_new.writeref \leftarrow 1$
38: $fpentry\_new.LAT \leftarrow current\ time$
39: **end if**
40: Copy data from buffer to the $fpentry.dbaaddr$.
41: Continue for the next block.
42: **end for**

4.2.1.1 Modified-ARC

There are many cache replacement policies such as LRU, 2Q, MQ, LRU-K and ARC. Among these algorithms, ARC is considered to be a very effective cache replacement policy for workloads with the weak temporal locality. ARC considers recency as well as frequency of the workloads and adaptively manages content in the cache. ARC algorithm maintains

four LRU lists $T1$, $T2$, $B1$ and $B2$ similar to the lists shown in Fig. 4.4. In these lists, the Most Recently Used (MRU) entries are stored at the left and the Least Recently Used (LRU) entries are stored at the right. Among these lists, $T1$ and $T2$ maintain recently accessed contents of data blocks along with their metadata, and $B1$ and $B2$ maintain only the metadata of the evicted cache blocks. When a block is accessed for the first time, it is placed in the $T1$ list. If a block from $T1$ is accessed for the second time, that block is shifted to the MRU position in $T2$. Evicted entries from $T1$ are moved to MRU position in $B1$ and similarly evicted entries of $T2$ are moved to MRU position in $B2$. Sizes of $T1$ and $T2$ are adjusted adaptively and controlled by the parameter p, (desirable size of $T1$, explained later in this section).

In this work, a Modified-ARC algorithm is proposed for content-based cache with deduplication. This Modified-ARC has $T1$, $B1$ and $B2$, which are organized as LRU lists, as that of ARC. However, $T2$ is organized differently based on additional parameters such as counts of write references and read references, the staying period in the cache and the last reference interval. These parameters are used to compute the popularity of a cache block.

The cache may receive read-intensive references, write-intensive references or mixed references at a particular interval of time. Instead of assigning static weightage to these references, dynamically computed weightage, which is more suitable, is applied. Apart from this, there may be cache blocks with high reference counts but not referenced for a long time. Such blocks may create hindrance for other recent popular blocks with fewer reference counts, by occupying the cache space. So, the popularity of the blocks which have not been referenced for a long time is decreased based on the length of the idle staying period. The last interval during which, the entry is not referenced, is considered as an idle staying period.

At periodic intervals of time, the weightage of write references and read references for the overall cache is computed, which is used for determining the popularity of the cache blocks of $T2$. Suppose in an interval *i*, total number of read references is N_r^i and total number of write references is N_w^i, then the weight of read references wt_r and the weight of

write references wt_w, are computed as follows.

$$wt_r = \frac{N_r^i}{N_r^i + N_w^i} \tag{4.1}$$

$$wt_w = \frac{N_w^i}{N_r^i + N_w^i} \tag{4.2}$$

For a particular cache block, if the number of read references is N_r and write references is N_w then weighted frequency N_{ref}^{wt} is

$$N_{ref}^{wt} = wt_r * N_r + wt_r * N_w \tag{4.3}$$

Let t_c denotes the current time and t_l denotes the last access time, of a cache block, then its popularity is

$$Popularity = \frac{N_{ref}^{wt}}{t_c - t_l} \tag{4.4}$$

In order to differentiate between the most popular and the least popular cache blocks of $T2$, Insertion Point (IP) is considered as shown in Figure 4.4, which is proposed in [13]. In this diagram, the circled numbers indicate different scenarios. When a block is referenced, which is not present in any of the lists $T1$, $T2$, $B1$ and $B2$ (scenario 1), then a new cache entry is created (if necessary, replacing an existing entry) and moved to the MRU position in $T1$. If the referenced block is found in $T1$ (scenario 2), its metadata is updated and moved to $T2$. If the referenced block is found in $T2$ (scenario 5), its metadata is updated and moved to the appropriate position in $T2$ (Described in the next paragraph). When the referenced block is found in $B1$ (scenario 3) or $B2$ (scenario 4), then the data is fetched from the disk and the corresponding entry with remembered metadata is moved to $T2$. While inserting an entry into $T1$ or $T2$, if the cache capacity exceeds its limit, an existing entry is replaced. If the entry being replaced is in $T1$, then the victim entry with only the metadata is moved to the MRU position in $B1$ (scenario 6). Similarly, if the entry being replaced is in $T2$, it is moved to the MRU position in $B2$ (scenario 7).

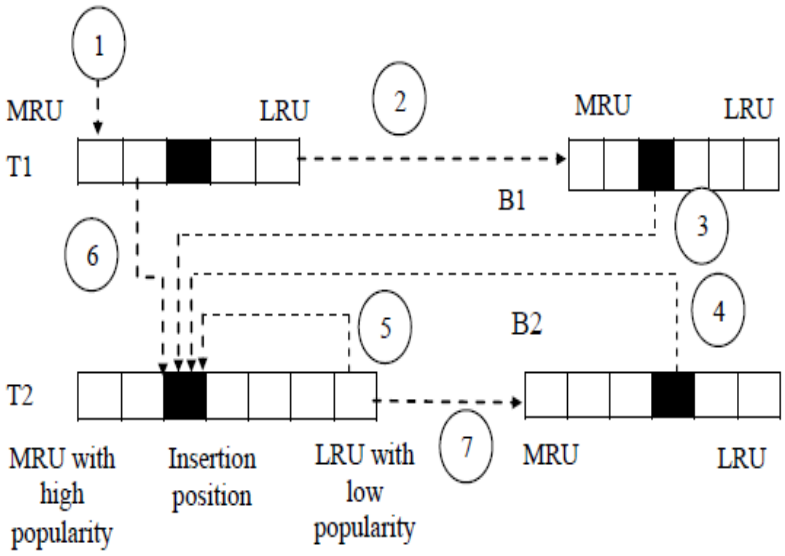


Figure 4.4: Modified-ARC algorithm

When a cache block is being moved to $T2$, two steps are followed. The first step is to compare the popularity of the current block with that of the block pointed by IP. If the popularity of the current block is higher than that of the block at IP, then it is moved to the MRU position in $T2$. Otherwise, it is inserted just after the IP. The second step is to adjust the cache size between $T1$ and $T2$, and it is adjusted adaptively based on the parameter p. Let c is the total capacity of cache, and p denotes the desirable size for $T1$, $p \in [0, c]$. When the current cache block is being moved from $B1$ to $T2$, equations 4.5 and 4.6 are used, and equations 4.7 and 4.8 are used when it is being moved from $B2$ to $T2$, to update the value of p. In these equations, $|T1|$, $|T2|$, $|B1|$, and $|B2|$ denote the sizes of $T1$, $T2$, $B1$, and $B2$ respectively. If the cache is full, then $|T1| + |T2| = c$, but the total cached items in $T1$, $T2$, $B1$, and $B2$ is no more than $2c$. Modified-ARC is explained in Algorithm 4.2.

$$p = \begin{cases} min\{p+1, c\}, |B1| \geq |B2| & (4.5) \\ min\{p + \dfrac{|B2|}{|B1|}, c\}, otherwise & (4.6) \end{cases}$$

$$p = \begin{cases} max\{p-1, 0\}, |B2| \geq |B1| & (4.7) \\ max\{p - \dfrac{|B1|}{|B2|}, 0\}, otherwise & (4.8) \end{cases}$$

Algorithm 4.2 Modified-ARC algorithm

Input: Request stream R_1, R_2 ... R_t

switch Search for R_i **do**
  case I: $R_i \in T1$
    Update metadata.
    if popularity(R_i) $\geq$ popularity(IP) **then**
      Move R_i to MRU position in $T2$.
    else
      Insert R_i just below IP.
    end if
  case II: $R_i \in T2$
    Update metadata.
    if R_i is in upper part **then**
      Move R_i to MRU position in $T2$.
    else
      if popularity(R_i) $\geq$ popularity(IP) **then**
        Move R_i to MRU position in $T2$.
      else
        Insert R_i just below IP.
      end if
    end if
  case III: $R_i \in B1$
    Read data from disk.
    Update metadata.
    if $|B1| >= |B2|$ **then**
      $k1 = 1$
    else
      $k1$ = $|B2|$ / $|B1|$.
    end if
    $p = min\{p + k1, c\}$
    ADJUSTCACHE(R_i, p)
    if popularity(R_i) $\geq$ popularity(IP) **then**
      Move R_i to MRU position in $T2$.
    else
      Insert R_i just below IP.
    end if

Algorithm 1.3 continuation

```
35:     case IV: R_i ∈ B2
36:         Read data from disk.
37:         Update metadata.
38:         if |B2| >= |B1| then
39:             k2 = 1
40:         else
41:             k2 = |B1| / |B2|
42:         end if
43:         p = max{p − k2, 0}
44:         ADJUSTCACHE(R_i, p)
45:         if popularity(R_i) ≥ popularity(IP) then
46:             Move R_i to MRU position in T2.
47:         else
48:             Insert R_i just below IP.
49:         end if
50:     case V: R_i ∉ T1 ∪ T2 ∪ B1 ∪ B2
51:         fp ← MD − 5(R_i)
52:         ADJUSTCACHE(R_i, p)
53:         Move R_i to MRU position in T1.
54:         Update metadata.
55: procedure ADJUSTCACHE(R_i, p)
56:     if |T1| + |T2| = c then
57:         if |T1| + |B1| ≥ c then
58:             if |T1| < c then
59:                 Evict LRU from B1.
60:                 REPLACE(R_i, p)
61:             else
62:                 if |B1| > 0 then
63:                     Evict LRU entry from B1.
64:                     Evict LRU entry from T1 and
65:                       move to MRU position in B1.
66:                 else
67:                     Evict LRU entry from T1 and
68:                       move to MRU position in B1.
69:                 end if
70:             end if
```

Algorithm 1.3 continuation

```
        else
            if |B1| + |B2| ≥ c then
                Evict LRU entry from B2.
                REPLACE(R_i, p)
            end if
        end if
    end if
end procedure
procedure REPLACE(R_i, p)
    if |T1| ≠ 0 and |T1| > p or R_i ∈ B2 and |T1| = p then
        Evict LRU block in T1 and
            move to MRU position in B1.
    else
        Evict LRU block in T2 and
            move to MRU position in B2.
    end if
end procedure
```

4.3 Experimental results

Prototype of the HDS, full deduplication system and native (without deduplication) systems are implemented in simulation environment under the Linux operating system running on Intel i7 processor based system. In all of these systems LRU, ARC and Modified-ARC cache replacement policies have been incorporated. In order to drive the simulation standard I/O traces taken from three production systems at FIU are used as input. These traces include the I/O requests generated by the virtual machines running web server (Web), file server (Home) and email server (Mail) [22], for a duration of 21 days. Trace statistics namely total I/O size, working set size, write-to-read ratio and unique data are given in Table 4.2. All three systems are executed with different cache replacement policies using the three FIU input traces. Experiments are conducted by varying the cache size from 10% to 80% of the total working set size for Web and Home datasets. For mail dataset, the cache size is varied from 2% to 20% of the working set size, because of its high duplicate I/O

requests and duplicate content. Metadata cache size is reserved at 4% of the total cache size, for all of the traces.

Performance of proposed HDS is compared with full deduplication and native systems, with different cache replacement policies, by varying cache sizes. The parameters used for the comparison are (i) hit ratio, (ii) average read response time per 4 KB block (iii) average write response time per 4 KB block (iv) normalized effective writes performed, (v) metadata overhead, in terms of number of metadata blocks accessed, per data block, (vi) write requests eliminated and (vii) average overhead for read and write requests.

Table 4.2: Trace statistics

	Mail	Web	Home
Total requests	460334027	14294158	17836701
Read requests	51348252	3116456	726464
Write requests	408985775	11177702	17110237
Total LBAs	14741706	549174	36340810
Duplicate data blocks	2110399	104870	9237083
Unique data blocks	12631307	444304	27103727
Working set size (KB)	58966824	2196696	145363240
% of duplicate data blocks (excluding multiple writes to the same block)	14.32	19.1	18.71

Hit ratio is considered as a major performance parameter for cache replacement policies. It is expressed as the percentage of total references found in the cache. Hit ratios for Web, Mail and Home datasets are shown in Figures 4.5, 4.6 and 4.7 respectively for the native system, full deduplication system and HDS. Modified-ARC and ARC outperform LRU for all the cases. For smaller cache sizes, the hit ratio with Modified-ARC is slightly improved, compared to ARC for all three systems. Hit ratio is nearly the same for both of the Modified-ARC and ARC, for all systems at 60% cache size for Web and Home datasets and 16% cache size for Mail dataset. High hit ratios for Web, Mail and Home datasets are

observed at cache size 60%, 16% and 60% respectively. However, hit ratio improvement is negligible when the cache size is increased beyond these limits. As cache size increases, highly accessed blocks are populated in the cache for any replacement policy, and the differences in effectiveness of these policies vanish. It can be observed that, for Home dataset, the hit ratio is poor for all of the policies. This is due to the poor data

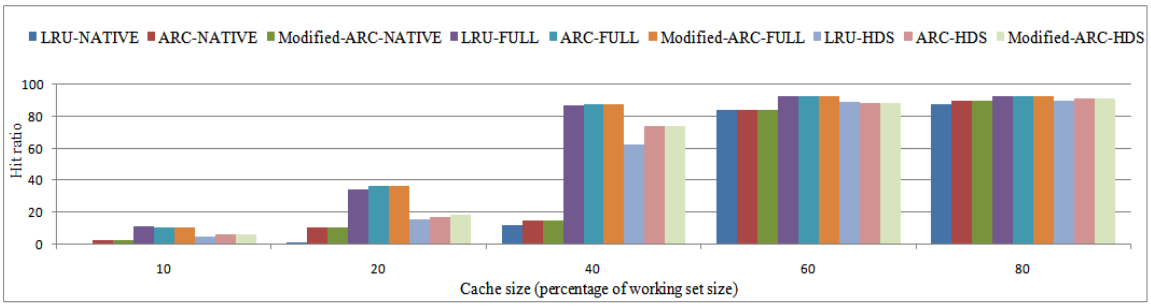


Figure 4.5: Web dataset hit ratio

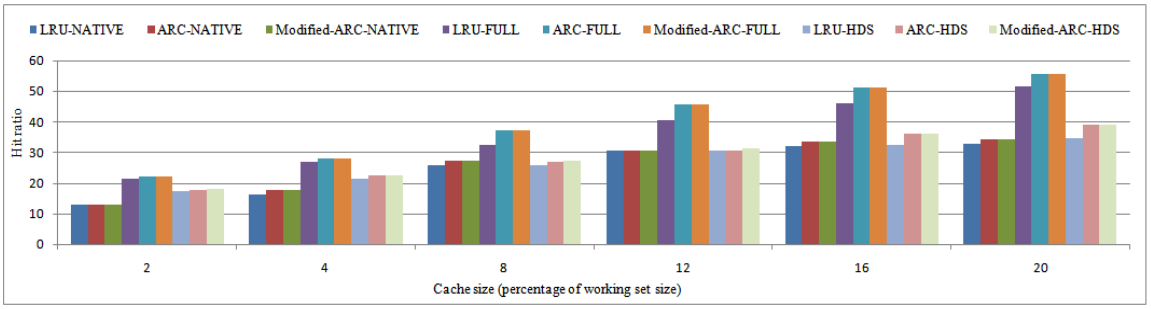


Figure 4.6: Mail dataset hit ratio

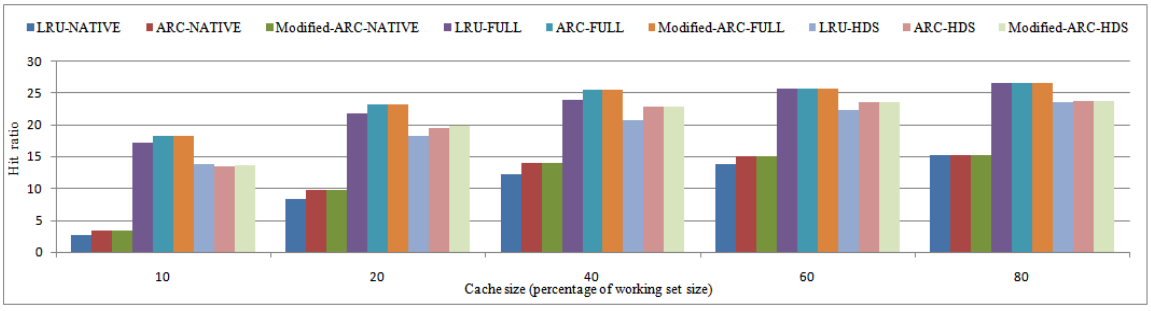


Figure 4.7: Home dataset hit ratio

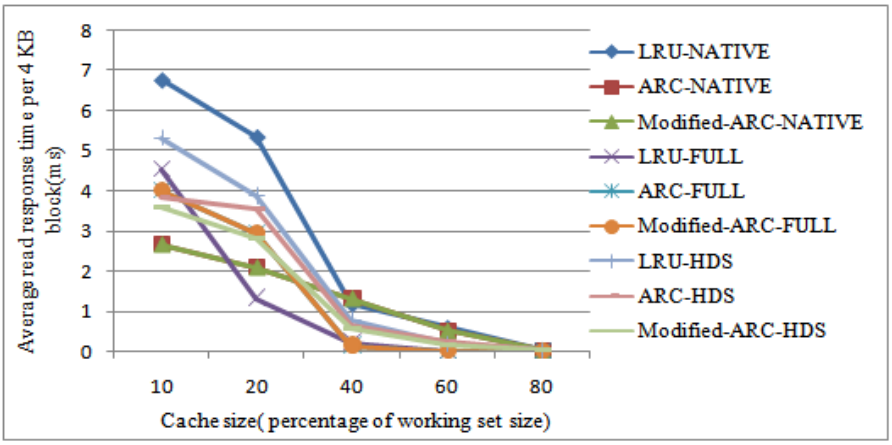
locality of the Home dataset.

Read response time and write response time determine the suitability of the proposed system to be used as a primary inline deduplication system. Timing statistics about read and write response times for different sizes of requests (4 KB to 1024 KB) are measured and details of standard 4 KB size requests are presented in this work. The average read

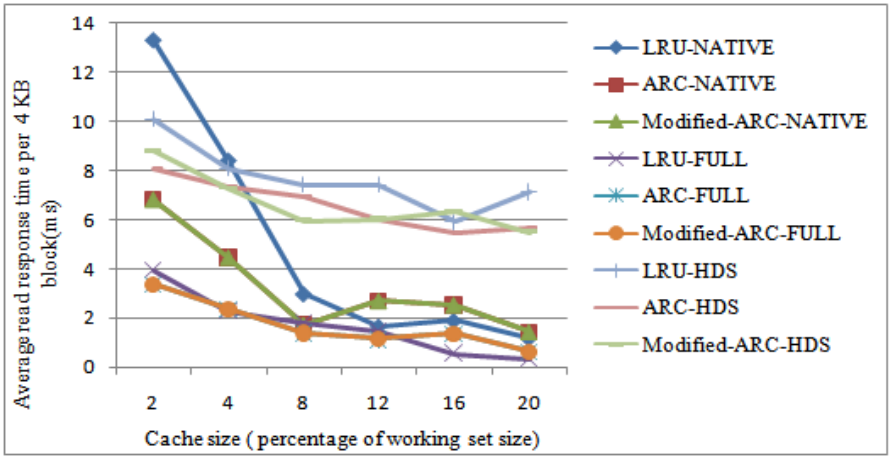
response time per 4 KB block is shown in Figures 4.8a, 4.8b and 4.8c for Web, Mail and Home datasets respectively. It can be observed that as the cache size is increased, average read I/O time per block decreases with Modified-ARC and ARC compared to LRU for all systems and all datasets. This is due to the improved replacement policy. The low improvement for the Home dataset compared to other datasets can be attributed to the smaller 512 byte block size, used in Home dataset. This is also a natural consequence of decreased hit ratio for Home datasets. Standard deviation of read response time is shown in Figures 4.9a, 4.9b and 4.9c for Web, Mail and Home datasets respectively. Similarly, the average write response time per 4 KB block is shown in Figures 4.10a, 4.10b and 4.10c for Web, Mail and Home datasets respectively. It can be observed that as cache size is increased, average write response time per block decreases with Modified-ARC and ARC compared to LRU for all systems and all datasets. Standard deviation of write response time is shown in Figures 4.11a, 4.11b and 4.11c for Web, Mail and Home datasets respectively.

Normalized effective writes is used as a parameter to measure the performance of deduplication and caching systems. Overwriting the blocks and writing duplicate content can be handled by the caching and deduplication techniques respectively. Actually performed writes at the disk system is a good measure of the effectiveness of deduplication and caching systems. Effective disk writes are counted and normalization is done by subtracting minimum effective write counts for each input trace. Normalization is done, because of the high range of the measured effective write counts. Normalized effective writes performed is shown in Figures 4.12a, 4.12b and 4.12c for Web, Mail and Home datasets respectively. It can be observed that effective writes performed, is decreased with modified-ARC and ARC, compared to LRU, for smaller cache sizes for all datasets and for all systems. But for larger cache sizes, the difference is negligible for all cases. Reduction in effective writes performed is poor, for Home dataset, due to the less duplicate content.

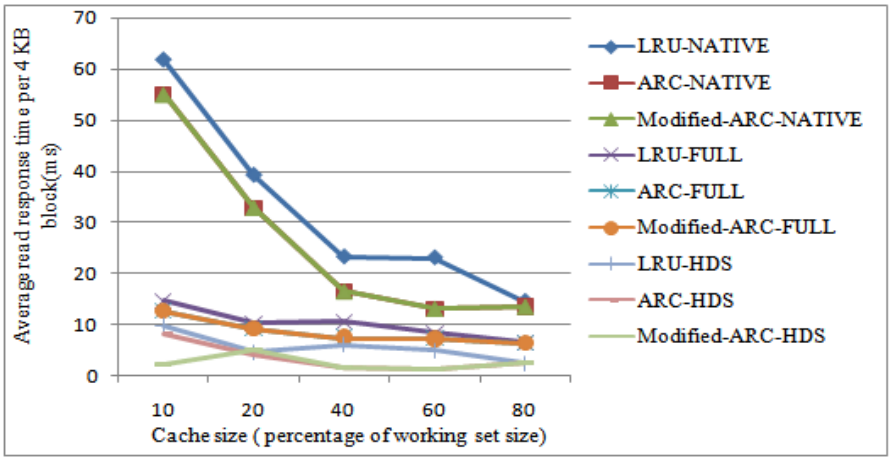
The effect of Insertion Position (IP) in Modified-ARC, on average I/O access time and effective write count, is studied using Mail dataset. For this experiment, the cache size is set to 2% of the working set size and insertion position has been varied from 10% to 98%. Insertion position partitions the $T2$ list into left IP% and right (100-IP)% of entries. Average I/O access time and normalized effective writes performed for the native system,



(a) Web dataset

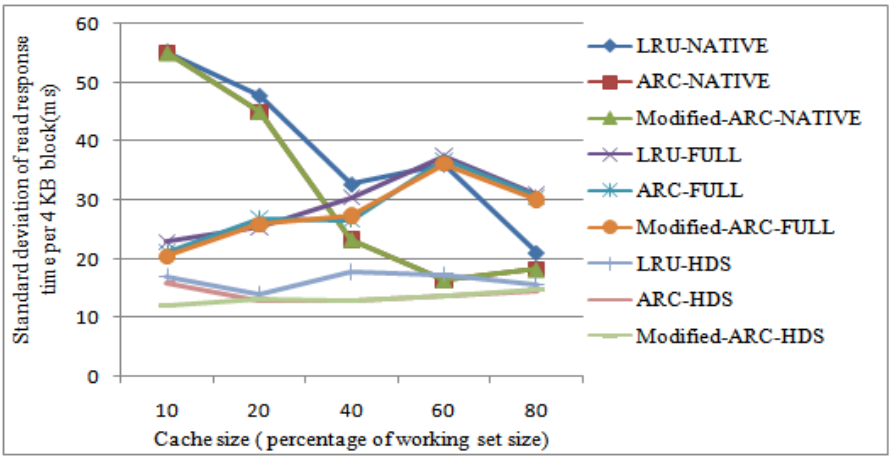


(b) Mail dataset

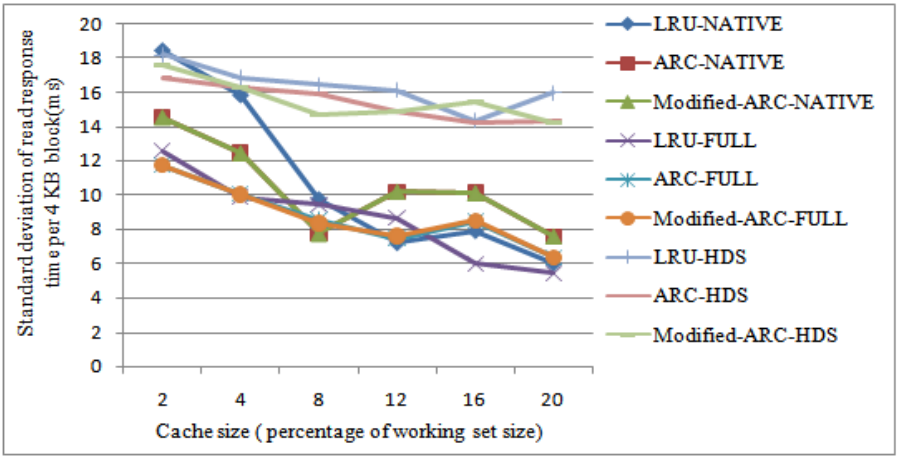


(c) Home dataset

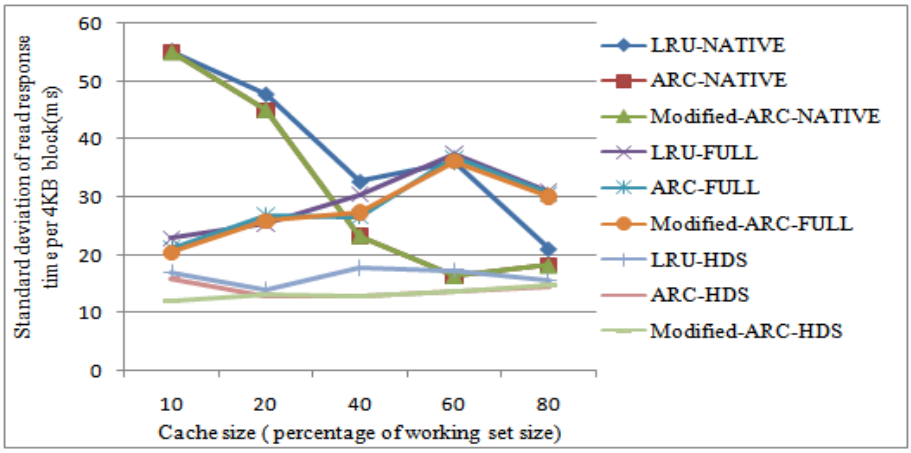
Figure 4.8: Average read response time



(a) Web dataset

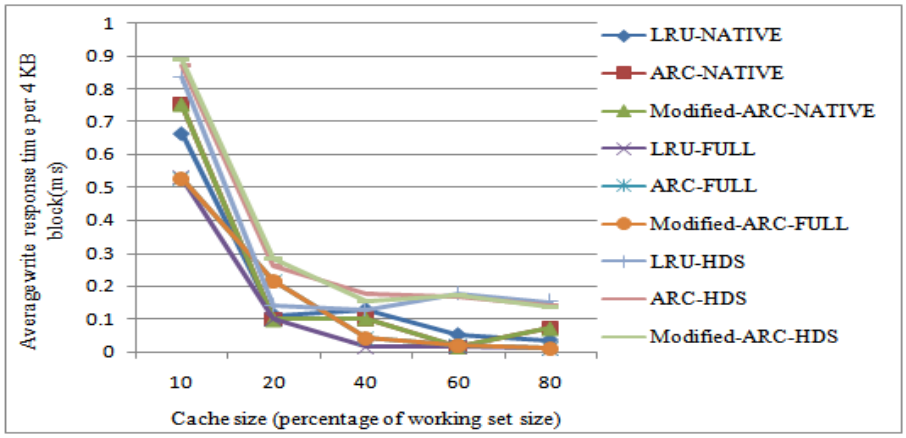


(b) Mail dataset

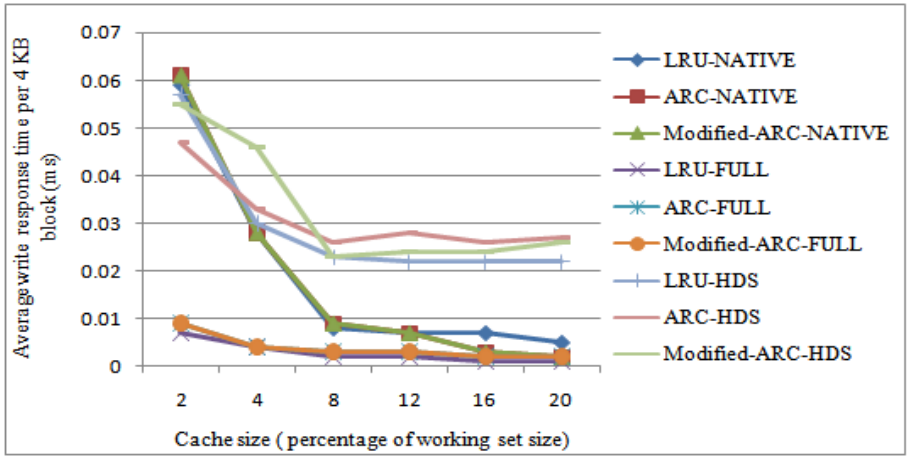


(c) Home dataset

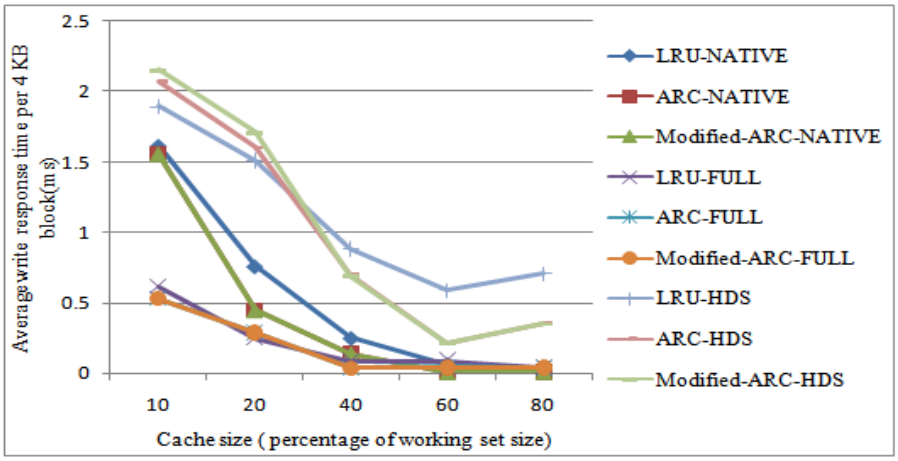
Figure 4.9: Standard deviation of read response time per 4 KB block (ms)



(a) Web dataset

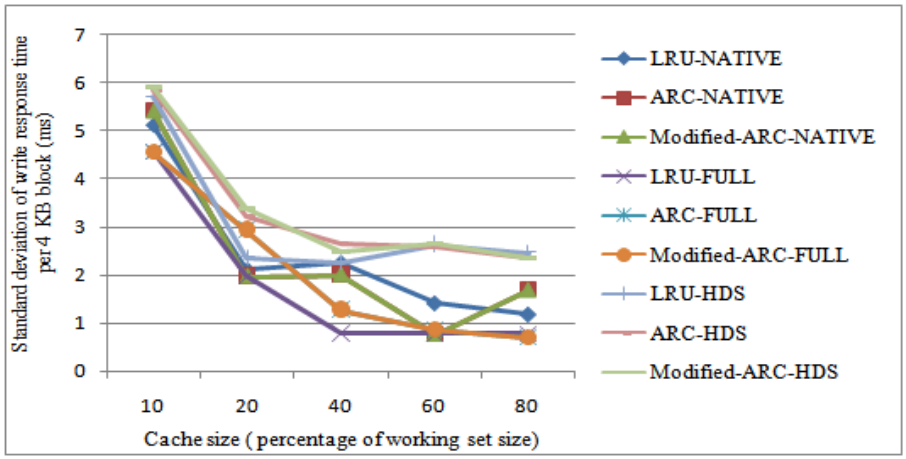


(b) Mail dataset

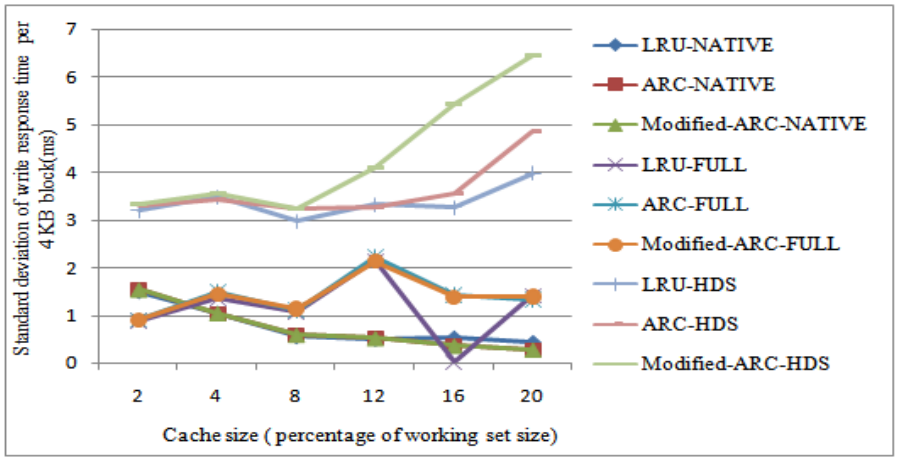


(c) Home dataset

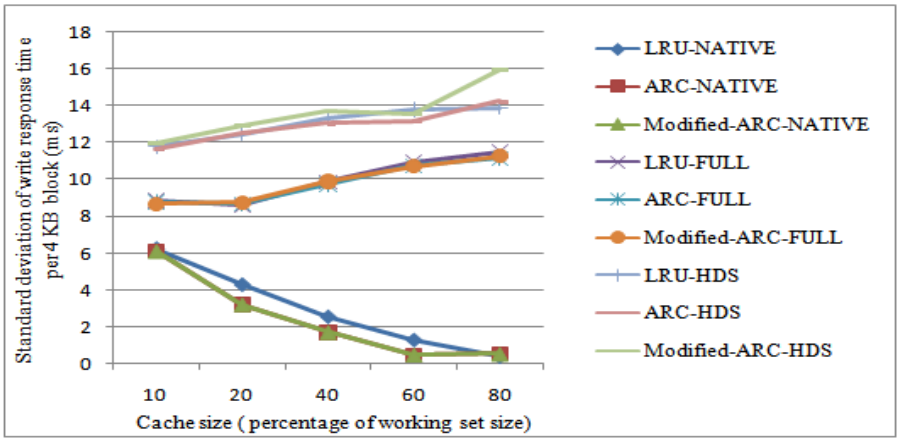
Figure 4.10: Average write response time



(a) Web dataset

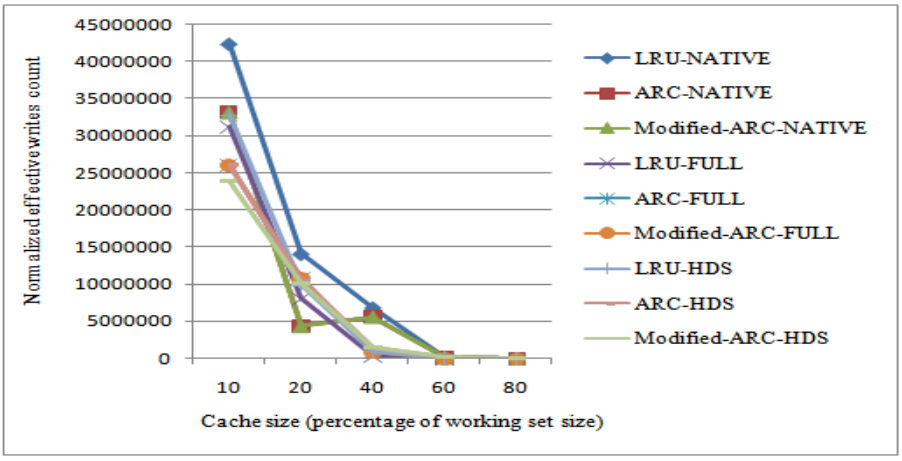


(b) Mail dataset

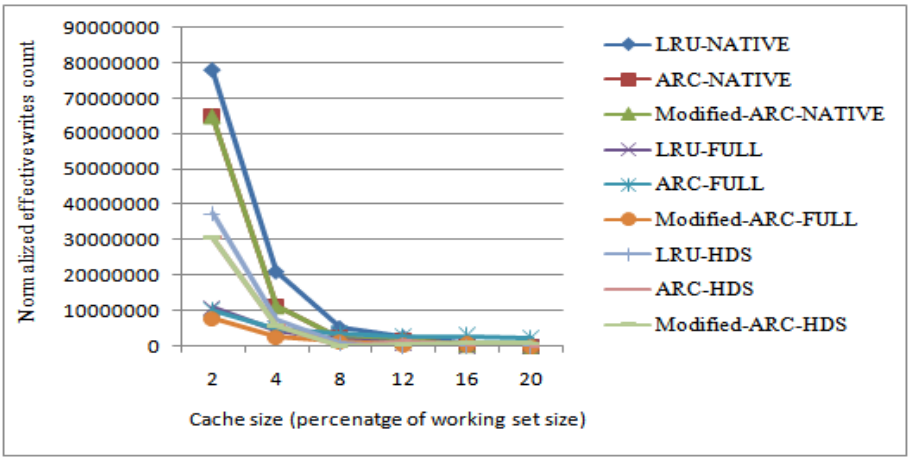


(c) Home dataset

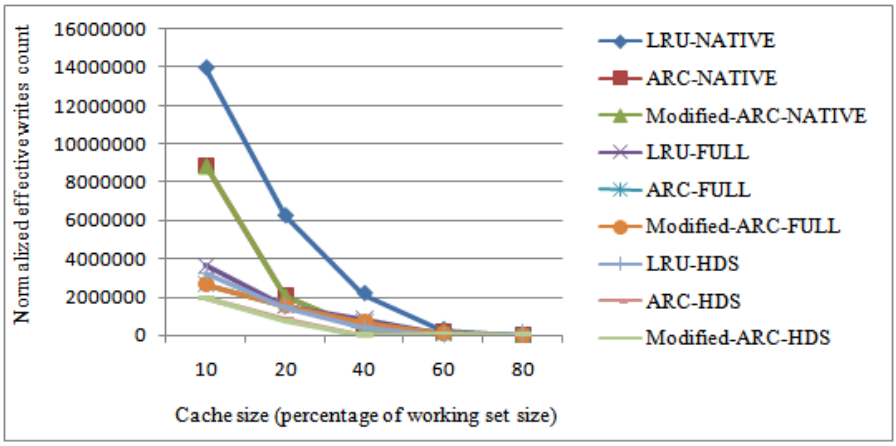
Figure 4.11: Standard deviation of write response time per 4 KB block (ms)



(a) Web dataset

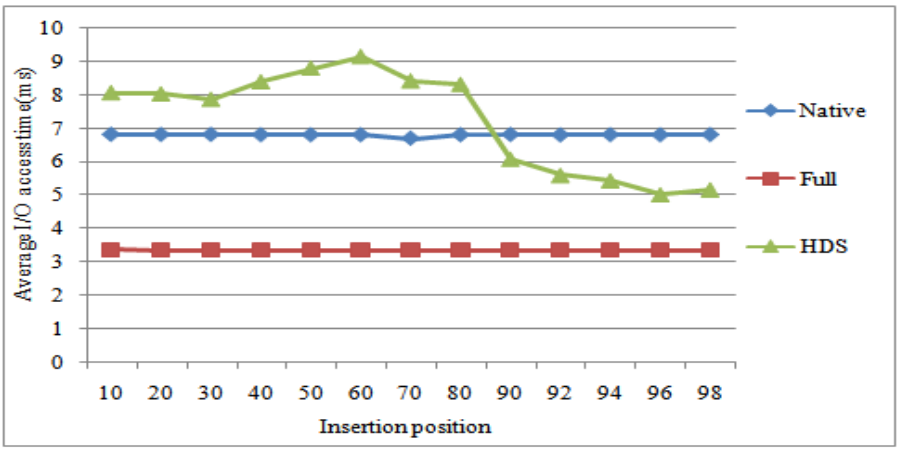


(b) Mail dataset

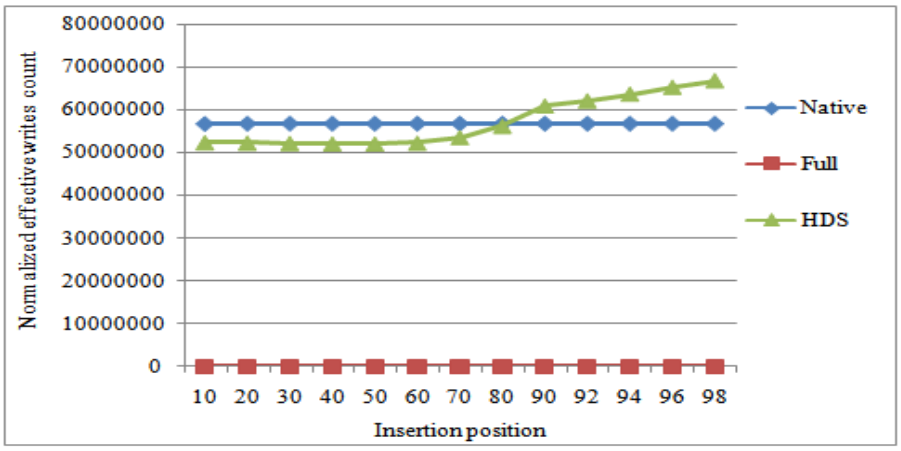


(c) Home dataset

Figure 4.12: Normalized effective write count



(a) Average I/O access time



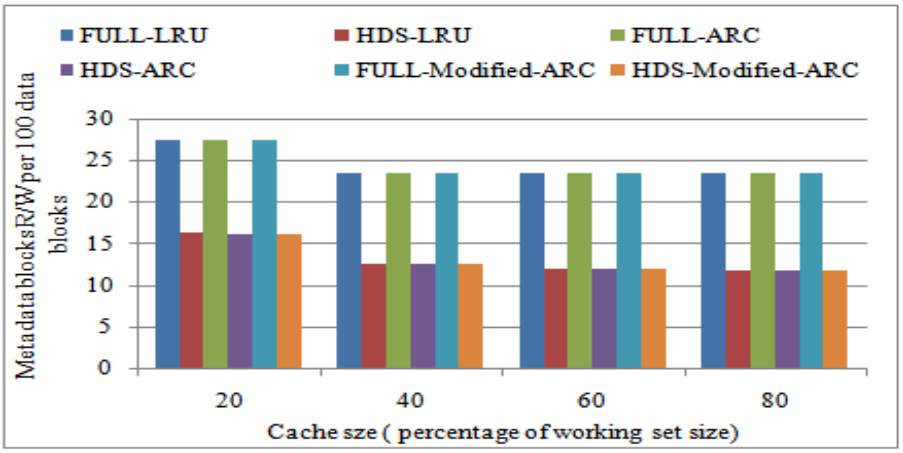
(b) Normalized effective writes

Figure 4.13: Effect of insertion position

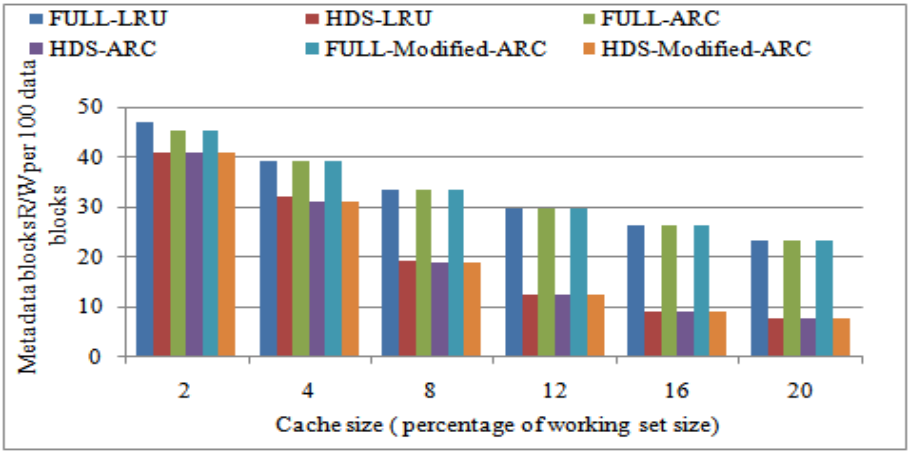
full deduplication system and HDS, are shown in Figures 4.13a and 4.13b respectively. As insertion position varies from 10% to 60%, average I/O access time and effective write count decreases. After optimal insertion position at 60%, there is an increase in both of the average I/O access time and effective write count, for all three systems. When insertion position is close to the MRU position, the algorithm behaves like LRU, by inserting the recently referenced entries either at the MRU position or close to the MRU position. However, if the insertion position is close to the LRU position, recently referenced entries with less reference counts may be inserted near to the LRU position, which leads to the eviction of such entries. Effectively less weightage is given to recent references. Hence insertion position at either extreme end leads to degraded performance of the Modified-ARC.

Metadata access overhead has a major impact on the performance of deduplication system. In the context of HDS, to study the effect of applying different cache replacement policies on metadata access overhead, an experiment is conducted to populate the cache with LRU, ARC and Modified-ARC by varying cache size. Overhead of metadata accesses is measured in two different ways by counting the reads and writes at the storage devices (both data disk and metadata disk). In the first method, using these measured values average number of metadata blocks read/written per 100 data blocks read/written is computed. In the second method, while processing the read/write requests inline, the actually performed metadata read/write operations are counted, separately for read and write operations. This gives inline overhead per data block read and written. Average numbers of metadata blocks read/written are shown in Figures 4.14a, 4.14b and 4.14c for Web, Mail and Home datasets respectively. It can be observed that metadata overhead of HDS is much lower than that of full deduplication system. The overhead values are proving the applicability of the proposed system for inline deduplication. Metadata read and write inline overhead per data block read and write operation for Web, Mail and Home dataset with respect to cache replacement policies at the specified cache size for datasets is given in Table 4.3. It is negligible value for HDS, for all input datasets. The values in the table are rounded upto four decimal places.

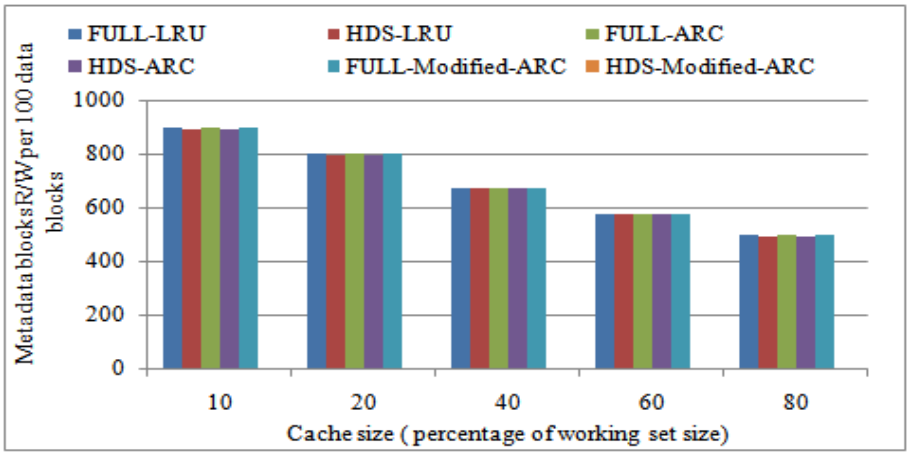
In order to measure the total writes eliminated, actual number of writes performed at the disk system are counted. For a given trace, total number of writes issued is also counted. The difference between these two values gives the number of writes eliminated. Caching absorbs the overwrites issued to the same data blocks and deduplication identifies and eliminates duplicate content written to the different data blocks. Write requests eliminated gives a measure of the performance of deduplication and caching systems. Percentage of write requests eliminated by applying deduplication along with different caching policies is shown in Figures 4.15a, 4.15b and 4.15c for Mail, Web and Home datasets respectively. The decrease in eliminated write requests in HDS is due to its selective deduplication.



(a) Web dataset

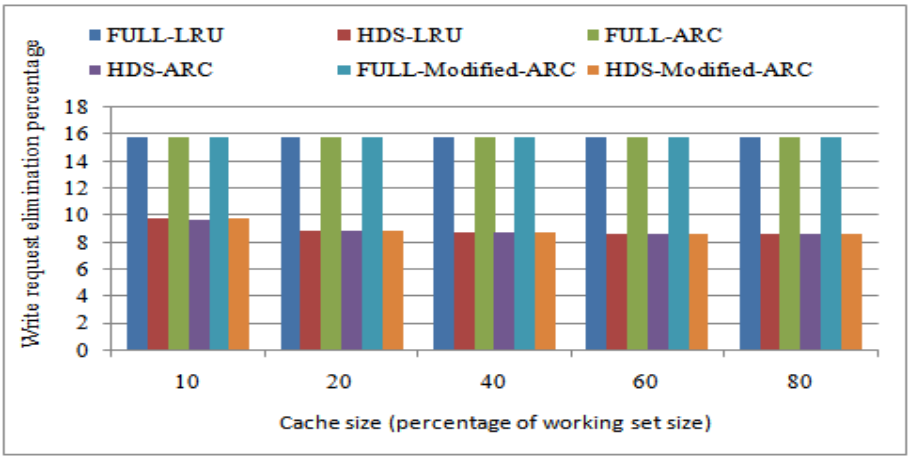


(b) Mail dataset

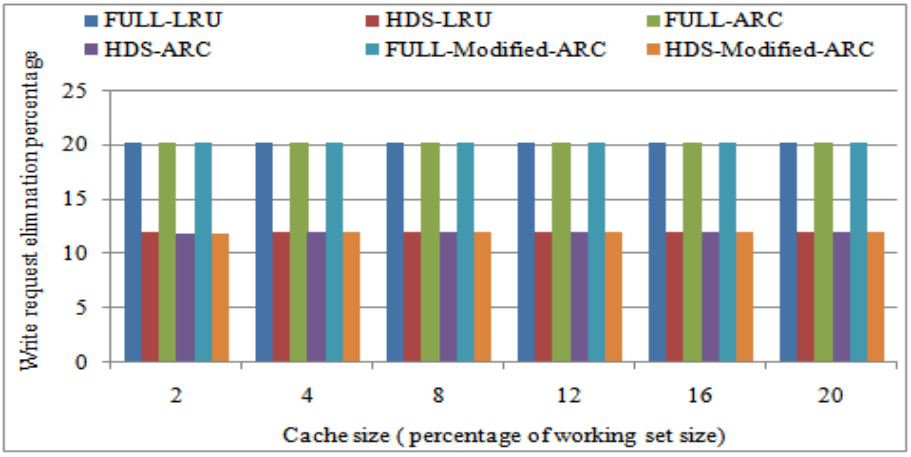


(c) Home dataset

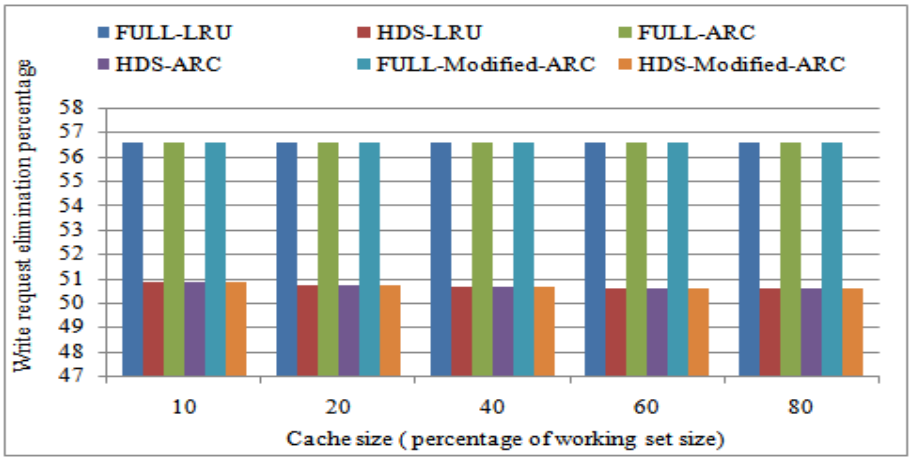
Figure 4.14: Metadata blocks R/W per 100 data blocks



(a) Web dataset



(b) Mail dataset



(c) Home dataset

Figure 4.15: Write request elimination percentage

Table 4.3: Metadata R/W inline overhead per data block R/W

	Web(40%)		Mail(20%)		Home(40%)	
Read overhead per data block read						
	HDS	FULL	HDS	FULL	HDS	FULL
LRU	0.0273	0.002	0.0412	0.1118	8.7757	7.5454
ARC	0.0038	0.002	0.0412	0.0309	8.7759	7.5434
Modified-ARC	0.0038	0.002	0.0412	0.1118	7.416	11.02
Write overhead per data block read						
LRU	0.4188	0	0	0	0.0108	0.5404
ARC	0.0064	0	0	0.0405	0.0106	0.4238
Modified-ARC	0.0064	0	0	0	0.0344	0.5075
Read overhead per data block write						
LRU	0.0133	0.0008	0.0314	0.18	4.5433	3.6539
ARC	0.0199	0.0008	0.0314	0.0022	4.5433	3.6513
Modified-ARC	0.0199	0.0008	0.0314	0.18	6.0477	2.2111
Read overhead per data block write						
LRU	0.0133	0	0	0	0.4493	1.2014
ARC	0.1296	0	0.0505	0.0647	0.4493	1.2295
Modified-ARC	0.1296	0	0.0505	0	0.4518	0.0002

4.4 Summary

Hybrid deduplication system with a content-based cache works at the block layer. It is mainly designed to enhance the performance of the deduplication system in the cloud environment, where the workloads have weak temporal locality. Disk-bottleneck problem arise due to random access pattern of deduplication metadata. To overcome this problem, similarity based indexing is used to keep metadata of similar segments together. In order to accelerate deduplication, a prototype of content-based data cache with Modified-ARC is implemented to populate the cache with unique data blocks. Modified-ARC considers weighted frequency and idle staying period, in addition to the recency of references for determining the popularity of the data blocks. It is found that Modified-ARC outperforms LRU. Experimental results have shown that HDS with content-based cache enhances sys-

tem performance. Effective writes performed is reduced with Modified-ARC compared to LRU and ARC. Overall I/O system performance is improved with Modified-ARC based HDS. In future, this cache replacement policy can be applied for increasing the durability of SSD.

Chapter 5

File Semantic Aware Primary Storage Deduplication System

The major platform for sharing various types of files, such as images, videos, emails, documents, backup files etc. is cloud-based storage. The different types of files have different levels of data redundancy. If a block-level deduplication system is used for low redundancy files such as video files, compressed files etc., oblivious of their data type, then negligible duplicate elimination with increased deduplication overhead can be observed. However, if the data redundancy level of a file is identified, this will help in utilizing the resources efficiently and achieve a significant amount of duplicate elimination. This chapter proposes an idea for categorizing files based on data redundancy and applying file type-specific deduplication.

The primary storage deduplication can be applied at either block level or file level. Block-level deduplication has only the content of the block whereas, file-level deduplication has both the file content and file semantics such as file type, size, access time and modification time etc. Irrespective of the level of applying deduplication, primary storage systems have the problems such as low data redundancy[4][5][6], latency sensitivity, and random access patterns [2][3]. In both of these approaches, deduplication can be applied directly on the I/O path of the primary storage systems. At both levels, applying deduplication raises problems such as extra latency on the I/O path, disk bottleneck and data fragmentation.

Deduplication is CPU intensive due to chunking and hashing as well as memory and I/O intensive due to index lookup. The direct application of deduplication on the I/O path causes an increase in latency. Deduplication metadata consists of information about fingerprints and the location of stored data blocks on the disks. The cryptographic fingerprints have random values and poor locality, due to which the disk is frequently accessed. This is known as the disk-bottleneck problem [16]. As metadata access overhead is increased, system performance is degraded. Another factor affecting deduplication performance is the criteria for duplicate elimination. There are two ways for duplicate data elimination namely - exact deduplication and near-exact deduplication. In the former approach, every duplicate block is eliminated which may result in data fragmentation. Whereas, in the latter approach duplicate data is eliminated selectively [8][14] to minimize the data fragmentation. Most of the recent research works have tried to address disk-bottleneck and data fragmentation problems at the block level. However, addressing these problems, deduplication systems that are aware of file semantics, can be found rarely.

File semantic aware deduplication research works can be seen in the context of either the secondary storage deduplication systems [18][20][76] or distributed primary storage deduplication systems [17][19]. Semantic-Aware Multi-tiered source deduplication framework (SAM) [18] is a backup deduplication system that is based on the file semantics. Initially, global file-level deduplication is conducted which is followed by local chunk level deduplication. SAM avoids deduplication of small files as well as compressed files. AA-Plus [76] is a backup deduplication system. In order to exploit spatial locality for read performance enhancement, the data chunks of the same application are stored together. AA-Dedup [20] is an application-aware backup deduplication system. It performs application-aware chunking, hashing and maintains application-aware index tables. File semantic aware secondary storage deduplication system works are the motivation to design file semantic aware centralized primary storage deduplication system.

File semantic aware primary storage deduplication system needs to consider file size and file type as main attributes. Many researchers Shemi et al. [5], Meyer et al. [6] and Jin et al. [91] have assessed the effect of file size and type on the performance of primary storage deduplication systems. These systems have domination of small size file accesses over

large size file accesses. As applying deduplication on small files is a resource-intensive task with less space-saving, most of the works [8][18] can be found applying deduplication only on large size files. However, some researchers [14] have found that though small file deduplication results in less space-saving, system performance can be improved. In the context of large size files, file type also has an important role in deduplication. Based on the file type, duplicate content level of the file and the possibility of change of content over a period of time can be predicted. Files such as text, document, backup and virtual machine images undergo frequent changes and have more data redundancy. Files such as video, audio, image and compressed types have low data redundancy and these files merely undergo changes. Few types of files' content redundancy can't be determined. Based on content redundancy, files can be partitioned as highly duplicate, low duplicate and unpredictable duplicate. It has been observed that data redundancy, across different types of files is negligible [5][20]. Deduplication among files of mismatched types results in increased deduplication overhead and less storage capacity saving. If deduplication metadata is maintained based on file size and file type, deduplication overhead can be reduced. Apart from file semantics, chunking that determines the duplicate identification level, affects the duplicate elimination ratio. Chunking can be applied at fixed-size or variable-size block level or at whole file-level. Though variable size chunking identifies more data redundancy, its application is not feasible in the primary storage system, due to its overhead. Between the fixed size and file-level chunking, the former identifies more data redundancy than the latter. Application of file-level chunking for high redundancy files reduces resource usage with decrease in the deduplication ratio. Similarly, application of block level deduplication for low redundancy files results in low deduplication ratio at the cost of high resource usage. However, the file type-specific deduplication strategy helps in reducing the deduplication overhead and achieves storage space-saving.

In this chapter, the File Aware DeDuplication system (FADD) is proposed. Files are categorized as small files and large files. Based on extensions, large files can be categorized broadly as highly duplicate (H) type, low duplicate (L) type and unpredictable duplicate (U) type. H and U type large files undergo segment level deduplication and L type files undergo file-level deduplication. If fixed-size chunking is applied for H and U type files,

the deduplication ratio will be improved. Whereas, whole file chunking for L type files decreases computational resource usage. Deduplication metadata of large files is maintained separately for each type and hash table is used for small files of all types, so that overall metadata overhead can be reduced.

The major contributions of this chapter are as follows:

- File categorization based on data redundancy.
- Similar segments identification and grouping into buckets, for H and U type large size files.
- Whole file deduplication for L type large size files.
- Efficient organization of small size files metadata into Hash table

The rest of the chapter is organized as follows. Section 5.1 gives detailed explanation on the design of the FADD system. Experimental results are presented in Section 5.2 and section 5.3 concludes the work.

5.1 System architecture

FADD is a file level centralized primary storage deduplication system. In order to perform deduplication, file semantics namely file size and file type are considered. Functional modules of the FADD system are shown in Figure 5.1. FADD system consists of four modules namely - file categorization, file pre-processing, deduplication and metadata management modules.

High-level workflow of the FADD system is given in Algorithm 5.1. Files are categorized as set of small size files (SF) and large size files (LF) based on size (line 1). If file size is less than or equal to 8 KB, it is included into SF set otherwise, included into LF set (line 6-9). Files enclosed under LF set are further categorized based on data redundancy into three categories as H (high redundancy) or U (unpredictable) redundancy) or L (low redundancy). Segment based deduplication is applied on files enclosed under H and U category (line 15). File level deduplication is applied on L type files (line 17). Whereas,

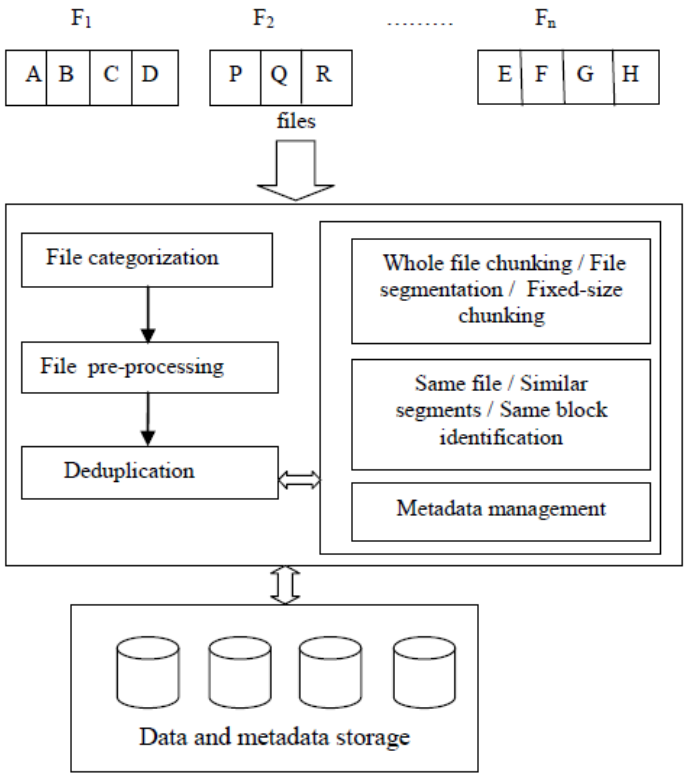


Figure 5.1: FADD system

Algorithm 5.1 Workflow of FADD system

Input: Sequence of files

Output: Deduplicated files and metadata

Let LF and SF denote the set of large size files and
    small size files respectively.
$LF \leftarrow \phi$
$SF \leftarrow \phi$
for each f in sequence of files **do**
    if $f.size >= 8$ KB **then**
        $LF \leftarrow LF \cup f$
    else
        $SF \leftarrow SF \cup f$
    end if
end for
for each f∈LF **do**
    Categorize the file f based on its extension, as H or U or L.

Algorithm 5.1 continuation

```
14:     if f ∈H or U  then
15:         Perform segment based deduplication.
16:     else
17:         Perform file level deduplication.
18:     end if
19: end for
20: for each f∈SF do
21:     Perform chunk level deduplication.
22: end for
```

chunk level deduplication is applied on all files enclosed under SF set (line 21).

5.1.1 File categorization module

This module is responsible for classifying the files based on size and type. Initially, file size is considered to filter out small size files from large size files. In this work, file sizes less than or equal to 8 KB are considered as small size files and remaining as large size files. Large size files are further categorized based on data redundancy. Files have data redundancy that may vary as high, low or unpredictable. Files such as text, document, backup and virtual machine images have more data redundancy and undergo frequent changes. Files such as video, audio, image and compressed files have low data redundancy which merely undergo changes. Files that have high duplicate data are treated as H type files. Whereas, files with low duplicate data are treated as L type files. Files without any extension or unknown extension are considered as U type files. Different types of files with few extensions enclosed under each category are given in Table 5.1.

Table 5.1: Categorization of large size files

File type	Files with extension
L type files	Video (.flv, .mp4, .m4v), Audio (.m4a, .mp3, .wma), Executable (.apk, .app, .exe), Bitmap image (.bmp, .gif, .jpg), Internet-related (.css, .html, .js), Encrypted (.dcf, .dco, .bin), Compressed (.rar, .pkg, .zip)
H type files	VM-related (.vmss, .vmtm, .vmx), Backup (.bak, .tmp), Disk image (.bin, .iso, .vcd), Configuration (.reg, .dll, .nfo), Database-related (.db, .mdb, .sql), Text (.rft, .doc, .sty), Document (.ppt, .pdf, .xml),
U type files	Miscellaneous (.ics, .msc), E-book (.epub, .mobi, .tr3), Email-related (.edb, .eml,.ics), Source code (.java, .sh, .py)

5.1.2 File pre-processing module

File pre-processing module performs the following subtasks - chunking, fingerprint computation and minhash computation. Small files of H, L and U type categories undergo fixed-size chunking and the MD-5 hash algorithm is applied to compute fingerprints of the chunks. Large files of L type undergo whole file chunking and the whole file hash is computed. Large files enclosed under H and U types are partitioned into segments of a fixed number of blocks (4 KB) and a fingerprint is computed for each block. Minimum fingerprint among all fingerprint values of a segment is taken as Representative IDentifier (RID) of that segment. All segments with the same RID are treated as similar segments.

5.1.3 Similar segments identification

A segment is a consecutive sequence of blocks. Broder's theorem [80] is applied to determine similar segments. Suppose, $S1$ and $S2$ are two segments. Let $H(S1)$ and $H(S2)$ denote sets of fingerprints of segments $S1$ and $S2$ respectively. Minimum fingerprint is identified for each segment and it is represented as $min(H(S1))$ and $min(H(S2))$ for

segment $S1$ and $S2$ respectively. H is chosen uniformly and at random from a minwise independent family of permutations. Broder's theorem states that similar segments share many similar data blocks and chances of having the same minhash are also high, which is equivalent to their Jaccard similarity coefficient as given in equation 5.1.

$$Pr[min(H(S1) = min(H(S1)] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{5.1}$$

FADD system identifies the segments of H and U type large files. Minhash is computed for each segment. This minhash is considered as the RID of the segment. A similarity bucket is used to store all similar segments. Each segment has to maintain information such as fingerprints of blocks, Physical Block Addresses (PBA) and their reference counts. These buckets are indexed using similarity-based indexing. RID of the segment is searched in a similarity-based indexing table to locate the bucket of similar segments. If a bucket exists, the segment is deduplicated by identifying the duplicate fingerprints in the bucket and the metadata is updated. Otherwise, after allocating a new bucket, metadata of the segment is added to it. Similarity-based segment indexing restricts index lookup to that particular bucket only, for the identification of duplicate data.

5.1.4 Metadata management

Deduplication metadata of the FADD system is maintained into three types of metadata structures - hash table for small files (H, L and U type), whole file hash table for large files of L type and, RID table and lists of similar segment buckets, for large files of H and U types. Fields of the metadata structures are shown in Figure 5.2.

Metadata of small files is maintained in a hash table as shown in Figure 5.2. Fingerprints of the blocks of small files are hashed to map to the slots in the hash table. Each slot consists of multiple entries to handle collisions. Whenever a slot is overflowed, linked additional slots are created at the end of the hash table. Each entry in a slot maintains information such as block number, fingerprint and reference count.

Indexing information of large files is maintained in two separate index tables - one for L type files and another for H and U type files. Metadata of large files of L type is

maintained in a whole file hash table 5.2b, which is organized as a B-tree. An entry in this table contains file id, whole file hash, size of the file and a pointer to the file recipe. The file recipe, which is maintained as a separate data structure, contains the data block allocation details. Metadata of large files of H and U type are stored in buckets which are indexed using the RID table as shown in Figure 5.2c. Each bucket maintains metadata of similar segments. Bucket contains information such as block number, fingerprint and reference counts of the blocks of all mapped segments as shown in Figure 5.2d. When a bucket overflows, additionally linked buckets are allocated.

Metadata of small files is maintained in the hash table to speed up index lookup. As small files are highly accessed compared to large files, in primary storage systems, the hash table helps to reduce the overhead of deduplication. Similarly, as large files of L type have negligible data redundancy compared to H and U type files, the overhead of deduplication is reduced by limiting the search for a single whole file hash value. Large files of H and U types have high data redundancy and the deduplication overhead is alleviated by using a similarity-based indexing approach.

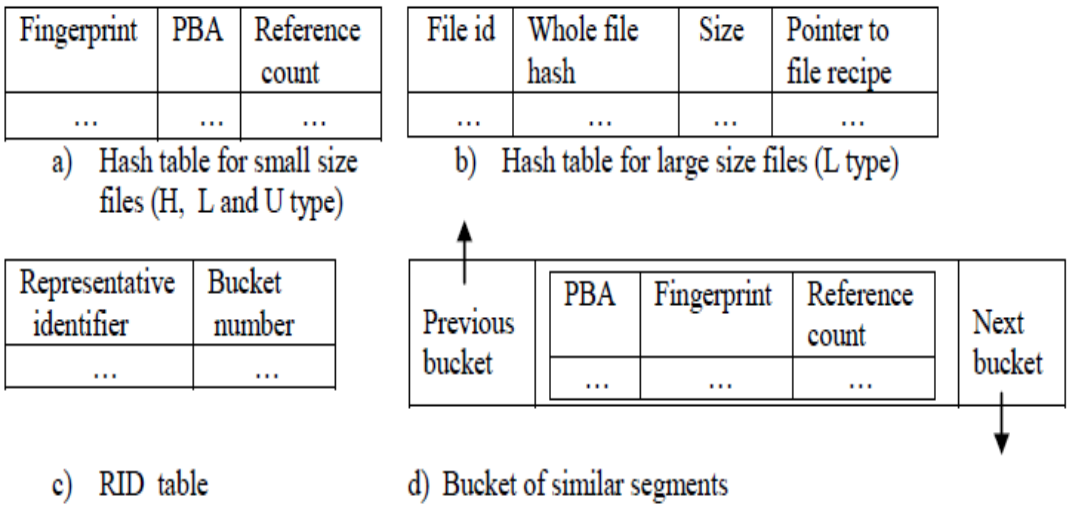


Figure 5.2: Metadata management

5.1.5 Deduplication

The usage of different approaches for small and large files minimizes the deduplication overhead for small files and improves the storage efficiency and I/O performance for large files. Hence, three types of deduplication approaches are followed - chunk based deduplication (small files of H, L and U type), whole file deduplication (large files of L type) and segment based deduplication (large files of H and U type). These approaches are explained in the following subsections along with respective algorithms. Table 5.2 describes the abbreviations used in the algorithms.

Table 5.2: Abbreviations used in the algorithms

Notation	**Description**
f	File
d	Data block
D	Sequence of data blocks
fp	Fingerprint of a block
s	Segment
S	Set of segments
F_s	Set of fingerprints of segment s
B	Bucket

5.1.5.1 Deduplication of small files of H, L and U type

Though access of small files is dominating over large files, applying deduplication achieves negligible storage capacity saving. It incurs more deduplication overhead if the same metadata structures are used for all types of files. However, most of the small files are duplicates and deduplication overhead is better than disk access for duplicate writes [14]. Small file deduplication helps to enhance I/O performance. Hash table is an appropriate data structure to maintain the metadata of small files, which enables faster access. As small files are highly accessed, small file deduplication results in frequent accessing of the corresponding metadata. Using the hash table, metadata operations can be performed faster. Thus, overhead incurred in small size file deduplication is minimized. Steps involved for the

deduplication of small size files (H, L and U type) are shown in Algorithm 5.2. Initially, fingerprint of a data block is computed (line 1). Next computed fingerprint is hashed to locate a slot in the hash table (line 3). If fingerprint is found, reference count is incremented (line 5). Otherwise, new entry is inserted in hash table slot and reference count is initialized to one. Data block is written to disk (line 10). Finally file recipe is added (line 13).

Algorithm 5.2 H, L and U type small size file deduplication

Input : File $f(LBA, PBAs, size)$

Output : Deduplicated file

1: Compute $fp(D) \leftarrow MD-5(D)$
2: **for** each fp **do**
3: Search for fp in the hash table.
4: **if** fp is found in the hash table **then**
5: $reference\_count++$, if required.
6: **else**
7: Insert new entry in hash table slot.
8: reference_count$\leftarrow 1$
9: Update metadata.
10: Write data block.
11: **end if**
12: **end for**
13: Add file recipe for *f(LBA, PBAs, size)*.

5.1.5.2 Deduplication of large files of H and U type

Large files of H and U type have more data redundancy and their deduplication improves storage efficiency. Segment based selective deduplication is applied to reduce the data fragmentation. Steps involved for the deduplication of large files of H and U type are shown in Algorithm 5.3. File is partitioned into segments (size 64 KB, 128 KB, 256 KB, 512 KB, 1024 KB) of fixed size blocks (4 KB) and a fingerprint is computed for each block (line 1). For each segment, the minimum fingerprint among all of its fingerprint values is taken as RID of that segment (line 5). In order to perform segment deduplication, the bucket of similar segments has to be identified. Deduplication module uses RID of the segment for searching the RID index to locate a bucket of similar segments (line 7). If

a bucket is not found, a new bucket is allocated, the new bucket number is entered in the RID index and the metadata of the segment is added to the bucket (line 9-12). If the bucket is already existing, then it is searched for the matching fingerprints. Deduplication module identifies and maps the duplicate blocks of the current segment to the blocks present in the bucket (line 14-23). New blocks are allocated for the non-duplicate content. In this process, selective deduplication (line 19-26) ensures that the mapping of the duplicate blocks to the existing blocks should not result in fragments of blocks less than a threshold count (In this work, the threshold count used is three). If it results in fragments that are less than the threshold size, duplicate data is not eliminated (line 20-22).

Algorithm 5.3 H and U type large file deduplication

Input : File $f(LBA, size, D)$

Output : Deduplicated file

1: Divide the file data D into segments of size 64 KB
2: (or 128 KB, 256 KB, 512 KB, 1024 KB) to get
3: a set of segments S.
4: **for** each segment $s \in S$ **do**
5: Perform fixed size chunking of segment s.
6: $F_s \leftarrow MD - 5(s)$
7: $MH \leftarrow Minimum(F_s)$
8: **if** $MH \notin RID$ index **then**
9: Assign new bucket number k for segment s.
10: Insert entry in RID index table with bucket number k.
11: Add metadata of the blocks to the B_k.
12: Write data blocks to storage.
13: **else**
14: Get corresponding bucket number k.
15: Obtain bucket B_k.
16: Search for fingerprints F_s in B_k.
17: Construct PBA segment(s).
18: Compute the lengths of PBA fragments.
19: **if** any fragment length $<$ fixed threshold **then**
20: Save the metadata of the segment
21: without eliminating duplicate block.
22: Write data blocks to the storage.

Algorithm 5.4 continuation

```
23:         else
24:             Add non-existing blocks' metadata.
25:             Update duplicate blocks' metadata.
26:             Write non-existing data blocks to the storage.
27:         end if
28:     end if
29: end for
30: Add file recipe for f(LBA, PBAs, size).
```

5.1.5.3 Deduplication of large files of L type

Large size files of L type have negligible data redundancy, compared to H and U type files. Applying block-based or segment-based deduplication for such types of files incurs unnecessary overhead and results in a very less saving of storage capacity. Hence, whole file deduplication is applied on L type large size files. Whole file deduplication works effectively by constraining the search for a single whole file hash value. While accessing the file using the file recipe, where the data blocks may be allocated in large contiguous chunks, high throughput can be achieved. Steps involved for the deduplication of large size files of L type are shown in Algorithm 5.4. Initially whole file hash fp is computed and searched in index table (line 1-2). If fp exists, its reference count is incremented (line 4). Otherwise, entry for whole file hash is inserted in table and reference count is initialized to 1. Along with this, respective metadata is also updated (line 6-8). Finally file recipe is added (line 10).

Algorithm 5.4 L type large file deduplication

Input : File $f(LBA, PBAs, size)$

Output : Deduplicated file

1: Compute whole file hash value fp.
2: Search whole file hash index.
3: **if** fp exists in the whole file hash index **then**
4: $reference\_count++$, if required.
5: **else**
6: Add new entry to whole file hash index.
7: $reference\_count \leftarrow 1$
8: Update metadata.
9: **end if**
10: Add file recipe $f(LBA, PBAs, size)$.

5.1.5.4 File read request processing

Though small and large file read requests are processed in a similar way, small read is given higher priority than the large read, to minimize the delay. Step by step procedure for file read requests processing is given in Algorithm 5.5. When a read request is received, using the file recipe corresponding PBAs are identified (line 1). Next buffer cache is searched for data of respective PBAs (line 2). If all data blocks are found, data is constructed and returned (line 4). Otherwise, a disk read request is issued for missing PBAs and retrieved from the disk and assembling of the cached block content into the application buffer is done and the data is returned (line 6-7).

Algorithm 5.5 File read processing

Input : Request $f(LBA, size)$

Output : File data blocks

```
1: Map LBAs to PBAs using file recipe.
2: Search for respective PBAs in the buffer cache.
3: if found then
4:     Assemble data and return.
5: else
6:     Issue disk read request to get data blocks.
7:     Construct data blocks into buffer and return data.
8: end if
```

5.2 Experimental results

Prototype of the FADD, HDS and Full deduplication systems are implemented and simulated under the Linux operating system running on an Intel i7 processor based system. Trace driven experiments are conducted to assess the system performance. Traces include standard I/O traces taken from two production systems at FIU and some locally collected data sets. The standard I/O traces consist of the I/O requests generated by the virtual machines running email server (Mail) and web server (Web) [22], for a duration of 21 days. In addition to this, the *Linux* dataset consisting of a collection of Linux kernel source code with version 5.x.y, *Book-ppt* data set consisting of books, ppts and documents, and *Video-image* data set consisting of video, audio and images, that are collected from desktops, are also used as inputs in the experiments. The datasets are categorized as H-type (Book-ppt and Linux), U-type (Mail and Web) and L-type (Video-image). For Mail and Web datasets (traces available without data), a file is identified as a sequence of read/write requests from the same process for the consecutive LBAs. Table 5.3 shows counts of total I/O requests, read requests, write requests, working set size, percentage of duplicate data, count of files, maximum, minimum and average file sizes for all the data sets used. Experiments are conducted by varying deduplication segment size from 16 to 256 blocks. In the present experiments, 20% of the working set size is used as the data cache and four percent of the

data cache size is reserved for metadata cache.

File type aware deduplication enhances overall I/O performance and storage efficiency. In the study, the parameters namely metadata overhead, overhead for inline processing of read request, the average length of a stored segment, storage space-saving, average read response time, average read throughput and average write throughput have been measured. Among these parameters (i) metadata overhead per data block is considered as the measure of disk-bottleneck, (ii) average length of a stored segment is taken as the measure of data fragmentation and (iii) response time is taken as a measure of request latency.

Table 5.3: Trace statistics

	Mail	Web	Linux	Book-ppt	Video-image
Total requests	460334027	14294158	86418389	4835460	21838832
Read requests	51348252	3116456	205236432	47644784	162206568
Write requests	408985775	11177702	86418389	9745351	55777601
Working set size (KB)	58966824	2196696	345673556	38981404	87355328
% of duplicate data	14.32	19.10	98.84	44.46	41.01
Count of files	855065	176982	22454899	80037	44392
Minimum file size (KB)	4	4	4	4	4
Maximum file size (KB)	37628	4096	16032	2978976	2926528
Average file size (KB)	69	12	15	487	5026

Overhead induced due to metadata accesses per data block is measured through two parameters namely the count of metadata blocks accessed while the data block is being read/written and the count of metadata operations (search, update, insert and delete) issued while the data block is being read/written.

First is the count of metadata blocks accessed while the data block is being read/written which is shown in Figure 5.3. Second is the count of metadata operations (search, update, insert and delete) issued while the data block is being read/written which is shown in Figure 5.4. It can be seen that the metadata overhead of FADD system for *Linux*, *Book-ppt* and *Video-image* data sets is much lower than that of the full deduplication system and HDS.

Metadata overhead in terms of metadata blocks read/written per data block read/written for FADD-256 in comparison with HDS, is reduced by 4.98%, 8.79%, 0.98%, and 28.37% for *Mail*, *Web*, *Book-ppt* and *Video-image* datasets respectively. For *Linux* dataset, reduction in the overhead is negligible for FADD-256 in comparison with HDS. Metadata over-

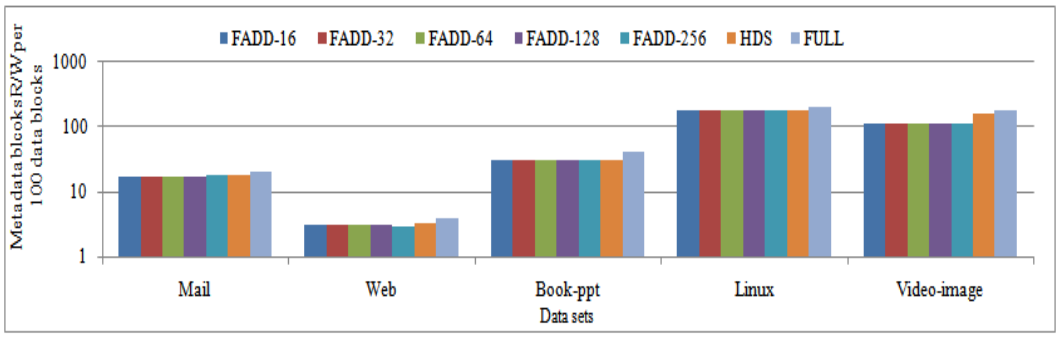


Figure 5.3: Metadata overhead (in terms of count of blocks of I/O)

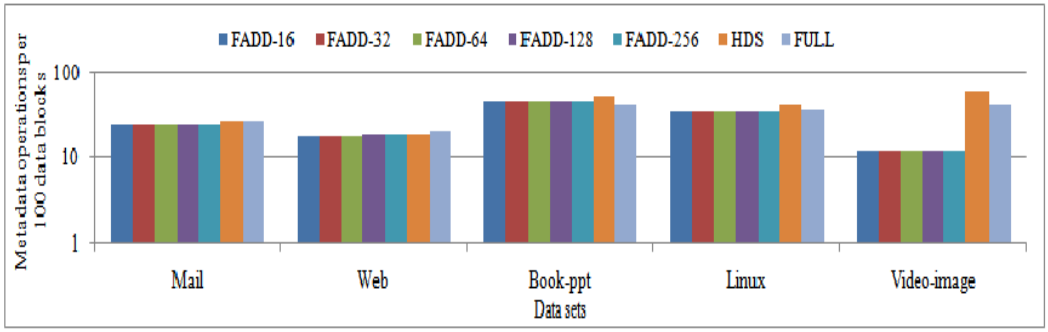


Figure 5.4: Metadata overhead (in terms of count of metadata operations)

head in terms of metadata operations performed per data block read/written for FADD-256 in comparison with HDS, is reduced by 9.76%, 4.37%, 13.62%, 18.14% and 79.75 for *Mail*, *Web*, *Book-ppt*, *Linux* and *Video-image* datasets respectively. Most of the files in *Linux* data set are small and deduplication metadata is stored in hashtables, which can be accessed faster with less number of operations. For *Video-image* data set, whole file hashing is used, and the search for duplicates requires only one lookup for the whole file hash. In addition to this as the duplicate content is more for *Linux*, *Book-ppt* and *Video-image* data sets (Table 5.3) in general the amount of metadata to be accessed is reduced on average. Metadata overhead for *Mail* and *Web* data sets is nearly equal to that of HDS and full deduplication systems. *Mail* and *Web* data sets are categorized as U type, and mostly the files are larger than 8 KB, so two disk accesses are required. First access for getting the bucket number from the RID index table and second access for fetching the bucket. The number of effective metadata accesses can be reduced further, by increasing the size of metadata cache.

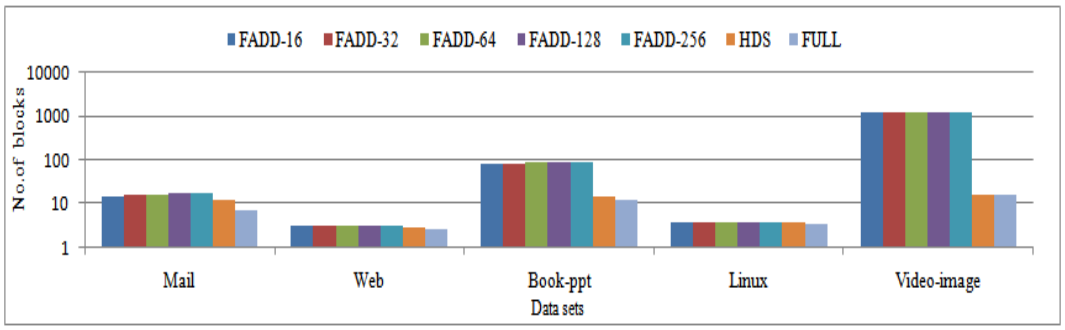


Figure 5.5: Average segment length

During deduplication, duplicate data blocks are eliminated which causes sequential data to scatter. As the FADD is performing selective deduplication, the average length of deduplicated data block segments is used as a metric for measuring data fragmentation. Larger the value of the average length of the segment, lesser the data fragmentation. Larger segments improve I/O performance for sequential access. Figure 5.5, shows the average lengths of segments for all data sets. Average segment length for FADD-256 in comparison with HDS, is increased by 31.46%, 10%, 83.91%, 3.71% and 98.67% for *Mail*, *Web*, *Book-ppt*, *Linux* and *Video-image* datasets respectively. The proposed FADD system has generated longer segments for all data sets, compared to the full deduplication system. It can be observed that within the FADD system as the deduplication segment size increases from 64 KB to 1 MB, the resultant average segment length is also increased.

Inline read overhead is measured for read operation as a number of metadata blocks being read/written per data block and it is given in Table 5.4. During experimentation, in order to measure the read overhead, hot cache is used for both data and metadata caches. It can be seen that FADD exhibits negligible inline read overhead. As increase in read latency is very minute, these results prove the applicability of FADD system for primary storage deduplication.

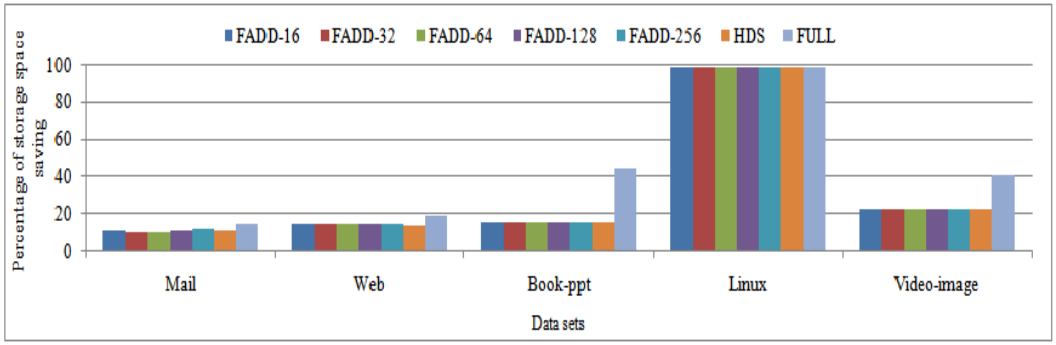


Figure 5.6: Storage space saving

Table 5.4: Read overhead per data block read in milliseconds

Dataset	FADD-256	HDS	FULL
Mail	0.0044	0.0054	0.006
Web	0.0059	0.0062	0.0068
Book-ppt	0.002	0.003	0.026
Linux	0.0108	0.0108	0.258
Video-image	0.0016	0.0016	0.2281

In order to measure storage space saved due to deduplication, two parameters are required - total effective writes issued (W) and actual number of data blocks stored after deduplication (D). Then

$$\text{Percentage of storage space saving} = \frac{(W - D)}{W} * 100 \tag{5.2}$$

Storage space saved due to deduplication is shown in Figure 5.6. The space-saving for the FADD system is close to that of the full deduplication system. For the FADD system, with increased segment lengths, it can be observed that the space-saving is slightly reduced.

Average read throughput is shown in Figure 5.7 and it is measured relatively. It can be observed that read throughput is increased with the FADD system compared to full deduplication and native systems. As data blocks are allocated in large contiguous chunks and data cache is used more efficiently, high throughput is achieved. In the context of *Linux* dataset most of the data is buffered and disk I/O is rarely required, so extremely high read throughput can be observed for deduplication systems compared to the native system.

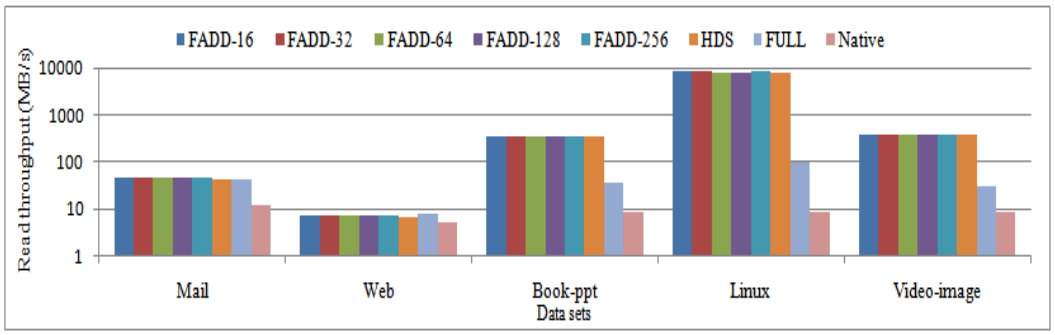


Figure 5.7: Read throughput

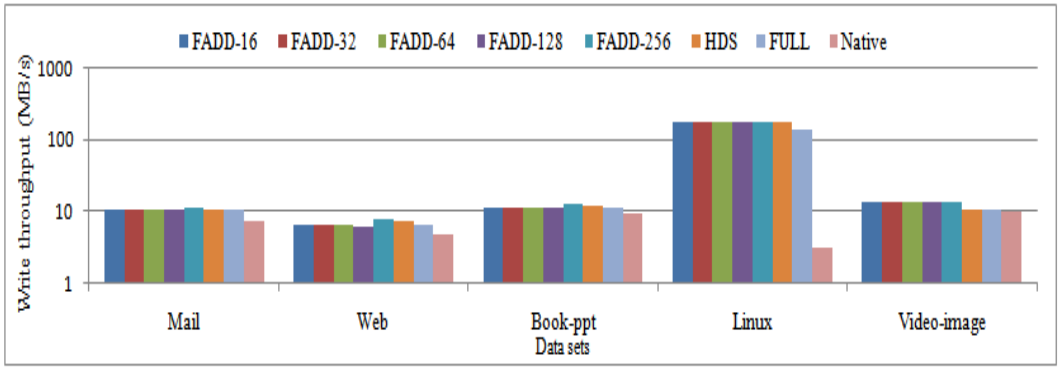


Figure 5.8: Write throughput

Average write throughput is shown in Figure 5.8. Write throughput is also measured relatively. It can be observed that write throughput is increased with the FADD system compared to full deduplication and native systems. As duplicates are removed and the blocks are allocated in large contiguous chunks, high throughput is achieved.

The average read response time per 4 KB block is shown in Table 5.5. It can be observed that the read response time is decreasing with FADD system as compared to the native system (without deduplication). This is due to the efficient utilization of the cache by eliminating the duplicate blocks. For *Linux* and *Book-ppt* data sets FADD system is showing better performance than even the full deduplication system, because of more reduction in metadata overhead. It can be concluded that when there is more duplicate content, FADD system can perform much better than HDS and full deduplication systems. The performance gain is more significant due to the elimination of duplicate data and the elimination of multiple writes to the same block/file within the buffer cache. FADD system uses delayed writes and deduplication is performed in the background, so the write response

times are not reported.

Table 5.5: Average read response time in milliseconds

Datasets	FADD-16	FADD-32	FADD-64	FADD-128	FADD-256	HDS	FULL	NATIVE
Mail	2.5151	2.519	2.5207	2.526	2.5319	2.5815	2.5373	2.8632
Web	2.3483	2.3475	2.3518	2.3539	2.358	2.4361	2.4585	4.3158
Book-ppt	0.0218	0.022	0.0221	0.0222	0.0221	0.0225	0.6049	3.7392
Linux	0.0046	0.0046	0.0047	0.0047	0.0047	0.0046	0.0072	3.5543
Video-image	0.6136	0.6136	0.6136	0.6136	0.6136	0.6611	0.6919	3.6136

5.3 Summary

File Aware Deduplication system works at the file level. It is mainly designed to enhance the performance of the deduplication system, where the workloads have weak temporal locality and have domination of small files accesses over large files accesses. Apart from this, large files have a different level of data redundancy. FADD system is considering file semantics such as size and type to categorize files as highly duplicate, low duplicate and unpredictable duplicate files. In order to overcome deduplication metadata access overhead, suitable separate metadata structures are used for each category. To improve read performance, segment level deduplication for high and unpredictable duplicate files and whole file deduplication for low duplicate files is applied. File type-specific deduplication saves resources and reduces deduplication overhead. In the experiments conducted, it is observed that the FADD system performed better than other systems. This work can be extended to implement a distributed deduplication system, consisting of multiple storage nodes and clients.

Chapter 6

File Aware Distributed Deduplication System in Cloud Environment

Centralized primary storage deduplication systems can store petabytes of data using deduplication technique. When the amount of data kept on these systems surpasses a certain limit, the performance of the system declines. Deduplication performance depends upon three factors - efficiency, scalability and throughput. Deduplication efficiency determines the capability of a system to identify and eliminate duplicate data. Scalability means the ability to handle a massive amount of data with consistent performance. Throughput refers to the rate at which the data is transferred to and from the system. Dedupli- cation system needs metadata indexing for identifying duplicates. However, the size of metadata scales linearly according to the system capacity. If metadata is maintained in memory, throughput is improved. But due to memory constraints, metadata has to be stored on disk. Though storing indexing metadata to disk would solve the scalability issue, the performance of the system is affected significantly. As data scales, it can not be stored on a single storage node and for building high capacity cloud storage systems distributed storage nodes are necessary. Hence, the need for distributed deduplication is raised to improve scalability and deduplication efficiency. The performance of a distributed deduplication system is determined by how well data is distributed to the server to achieve a high duplicate elimination ratio while maintaining load balance. If high duplicate data elimination is desired in distributed deduplication systems, all files of the same type must be routed to the same server,

causing a load imbalance problem. Duplicate data elimination should be compromised if load balance is to be accomplished. As a result, getting a high level of duplicate elimination with load balance is challenging. In this work, files are categorized as high duplicate, low duplicate and unpredictable duplicate based on levels of data redundancy. Files are assigned using a stateless and a stateful approach based on file type.

Distributed deduplication system consists of multiple data servers handling data from multiple clients simultaneously. Deduplication is CPU intensive due to chunking and hashing as well as I/O intensive due to index lookup. In a distributed environment, deduplication tasks can be shared among the clients and the data servers, in order to make the system scalable. Distributed deduplication system has to consider various issues such as the location of deduplication, splitting of deduplication tasks between the client and the data server, proper partitioning of data and load balancing. Deduplication location can be either the client or the data server. If the deduplication is performed at the client, duplicate data transfer can be avoided and the data servers are free from the load of deduplication tasks. But the duplicates among different clients cannot be eliminated. If the deduplication is performed at the data server, all clients transfer their entire data which may consume more network bandwidth. The data server eliminates the duplicate data among all the clients, which increases the load on the data server. Deduplication at the data server has to consider whether duplicate data elimination is to be performed within a data server or across all data servers. If deduplication is performed at each data server independently, unaware of the redundancy across the data servers, each one becomes deduplication information island [21]. If deduplication is performed across data servers, then a global index table has to be maintained. As all data servers have to access this index table for identifying duplicate data, it results in a bottleneck.

Another issue is the splitting of deduplication tasks between client and data server. Among deduplication tasks, data chunking and fingerprint computation are CPU intensive. Whereas, index lookup and duplicate eliminations are memory and I/O intensive tasks. Data chunking and fingerprint computation can be performed at the client and index lookup and duplicate elimination can be assigned to the data server. Partitioning of deduplication tasks between client and data server, accelerates the deduplication process and

also improves the scalability.

The next issue is the assignment of the data among data servers, to achieve maximum duplicate elimination with load balancing. Existing works on distributed deduplication systems perform data assignment either file type aware [17][21] or file type oblivious [15][59][61][67]. File type aware assignment gives a better deduplication ratio while file type oblivious assignment incurs less overhead. Another type of categorization of data assignment approach is either stateful approach or stateless approach. In a stateful approach, the data present at the data servers is considered while assigning the data among the data servers. In the stateless approach, the decision is taken purely based on only the data being assigned. In the former approach, load balancing can be achieved, whereas it is ignored in the latter approach.

Many studies have been conducted to understand the effect of file type and file size on deduplication. Jin et al. [91], Meyer et al. [6] and Shemi et al. [5] have conducted studies on files of different extensions to find duplicate ratio. Their studies have revealed that different file types have different levels of duplicate content that may vary from high to unknown. Apart from this, there is insignificant duplicate content exists across different file types. This has motivated many existing works to consider file type based deduplication for distributed deduplication.

In the cloud, multiple clients generate different types of data that includes multimedia files, text files, backup files, VM related files, system configuration files and compressed files etc. These files are categorized as H (highly duplicate content), L (Low duplicate content) and U (unpredictable duplicate content) type based on their data redundancy. If chunking and fingerprint computation is performed at the client and index lookup at the server, it helps in two ways. Firstly it distributes resource-intensive deduplication tasks between the client and the data server. Secondly, it avoids duplicate data transfer and thereby saving network bandwidth. In order to enhance the deduplication ratio and to maintain data similarity, a set of dedicated data servers for each file category are allocated. Data servers handling specific category performs file type-specific deduplication. File level deduplication is applied to L type files, as they have less duplicate content, and these files are assigned to data servers using stateless routing. H and U type files are divided into a

sequence of superchunks (each superchunk is a sequence of segments), which are routed using two level routing and at the data server, segment level deduplication is applied.

In this chapter, Distributed Deduplication System (DDS) is proposed to perform file type aware, hybrid i.e., stateful and stateless distributed deduplication. The main contributions of this chapter are as follow.

- Classification of files based on the percentage of duplicate content
- File type-specific allocation of data servers
- File type-specific deduplication strategy and routing approaches
- Probabilistic approach to select the suitable data server
- Similarity-based indexing at the data servers

The rest of the chapter is organized as follows. Section 6.1 gives detailed explanation on the design and implementation of DDS. Experimental results and evaluation are presented in Section 6.2. Finally Section 6.3 concludes the work.

6.1 Distributed deduplication system

Distributed Deduplication System (DDS) performs deduplication based on the type of file. This section gives detailed explanation of system architecture, indexing data structures, and workflow for file storage (L, H and U type files) and retrieval process.

6.1.1 System architecture

As shown in Figure 6.1, the DDS system consists of a set of clients, coordinator and a set of data servers. Clients transfer files for deduplication to data servers. The coordinator is responsible for maintaining the information of data servers. Data servers are performing local deduplication and storing the files received from clients. Detailed explanation of DDS architecture is given below.

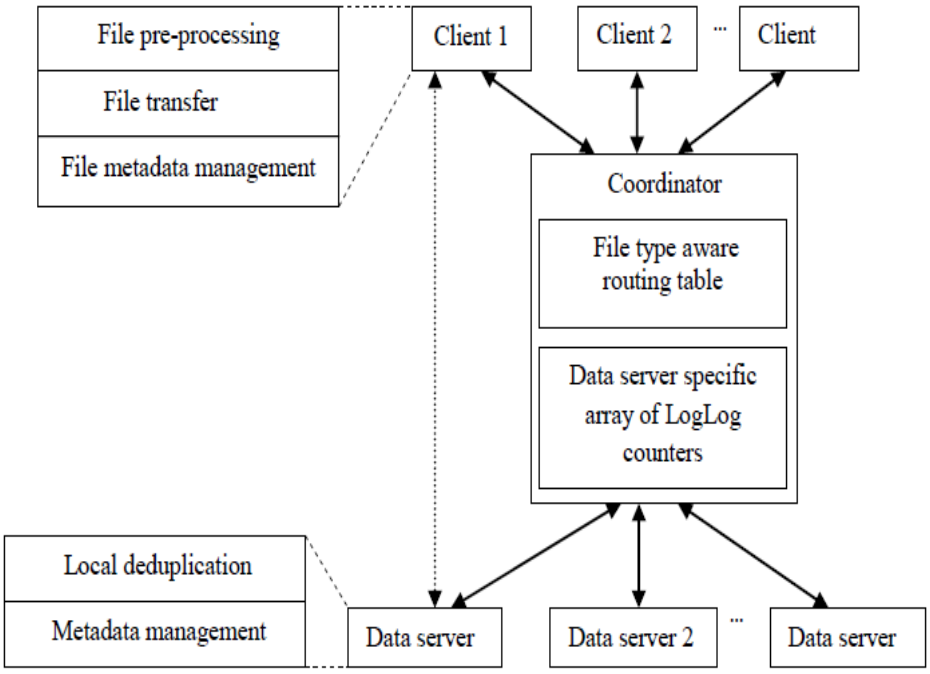


Figure 6.1: Distributed Deduplication System

6.1.1.1 Client

There are three functional modules in the client, namely file pre-processing module, file transfer module and file metadata management module.

File pre-processing module: This module is responsible to determine file category and chunking. The file is categorized as high duplicate (H), low duplicate (L) or unpredictable duplicate (U) based on the file extension. Few examples for each type are given below.

1. H type files: Virtual machine images, text, documents, power point presentation, portable document file, backup, configuration files, disk image, database-related etc.
2. L type files: Bitmap image, compressed, executable, temporary, video, audio etc.
3. U type files: Miscellaneous, script files, emails, ebook, source codes etc.

File transfer module: This module transfers file extension and whole file hash/minhashes of superchunks of the file to the coordinator and the coordinator replies indicating the suitable data server. For L type files, stateless routing is used and for H and U type files,

stateful routing is used.

File metadata management module: The client maintains a file recipe that is useful to retrieve a file from the data server.

6.1.1.2 Coordinator

It maintains information about the file extensions and a set of data server addresses handling that file type. LogLog counters that are required for the probabilistic estimation of cardinality (unique block count), are maintained for the data servers. Coordinator is responsible for determining the suitable data server for a given file/superchunk, for balancing the load.

6.1.1.3 Data server

Data Server consists of two modules namely metadata management and local deduplication.

Metadata management: Data servers that deduplicate H and U type files maintain segment 3.1.6.3 Read request processing For a given read request, LBAs are searched in LBA buffer cache. If all LBAs are found, data is constructed and sent. Otherwise, the LBAs are mapped to PBAs and PBA buffer cache is searched for the data. If found, data is constructed and sent. If data is not found in both LBA and PBA buffer caches, then disk read is issued to read data blocks into the buffer cache and the data is assembled and returned. Detailed steps for processing read requests are given in Algorithm 3.4. Algorithm 3.4 Read request processing Input: Request r(LBA address, size) from SR or LR queue 1: Search in the LBA buffer cache. 2: if found then 3: Construct data and return. 4: end if 5: Map LBA's to PBA's. 6: Search in the PBA buffer cache. 7: if found then 8: Construct data and return. 9: else 10: Generate disk read request to obtain data blocks. 11: Assemble data blocks into buffer and 12: return buffer data. 13: end if 3.2 Experimental results and evaluation Prototype of the HDS, Full deduplication system and native (without deduplication) sys- tems are implemented and simulated under the Linux operating system running on Intel i7 processor based system, with standard I/O traces taken from three production systems at FIU as input. The input includes the I/O requests generated by the virtual machines run- ning web server

(Web), file server (Home) and email server (Mail) [22], for a duration of 21 days. Table 3.2 shows counts of I/O requests, LBAs, duplicate blocks and unique blocks 55 Figure 5.6: Storage space saving Table 5.4: Read overhead per data block read in milliseconds Dataset FADD-256 HDS FULL Mail 0.0044 0.0054 0.006 Web 0.0059 0.0062 0.0068 Book-ppt 0.002 0.003 0.026 Linux 0.0108 0.0108 0.258 Video-image 0.0016 0.0016 0.2281 In order to measure storage space saved due to deduplication, two parameters are re- quired - total effective writes issued (W) and actual number of data blocks stored after deduplication (D). Then Percentage of storage space saving = (W D) W 100 (5.2) Storage space saved due to deduplication is shown in Figure 5.6. The space-saving for the FADD system is close to that of the full deduplication system. For the FADD system, with increased segment lengths, it can be observed that the space-saving is slightly reduced. Average read throughput is shown in Figure 5.7 and it is measured relatively. It can be observed that read throughput is increased with the FADD system compared to full deduplication and native systems. As data blocks are allocated in large contiguous chunks and data cache is used more efficiently, high throughput is achieved. In the context of Linux dataset most of the data is buffered and disk I/O is rarely required, so extremely high read throughput can be observed for deduplication systems compared to the native system. 123 Figure 5.7: Read throughput Figure 5.8: Write throughput Average write throughput is shown in Figure 5.8. Write throughput is also measured relatively. It can be observed that write throughput is increased with the FADD system compared to full deduplication and native systems. As duplicates are removed and the blocks are allocated in large contiguous chunks, high throughput is achieved. The average read response time per 4 KB block is shown in Table 5.5. It can be ob- served that the read response time is decreasing with FADD system as compared to the native system (without deduplication). This is due to the efficient utilization of the cache by eliminating the duplicate blocks. For Linux and Book-ppt data sets FADD system is showing better performance than even the full deduplication system, because of more reduction in metadata overhead. It can be concluded that when there is more duplicate content, FADD system can perform much better than HDS and full deduplication systems. The performance gain is more significant due to the elimination of duplicate data and the elimination of multiple writes to the same block/file within the buffer cache. FADD system uses

delayed writes and deduplication is performed in the background, so the write response 124 times are not reported. Table 5.5: Average read response time in milliseconds Datasets FADD-16 FADD-32 FADD-64 FADD-128 FADD-256 HDS FULL NATIVE Mail 2.5151 2.519 2.5207 2.526 2.5319 2.5815 2.5373 2.8632 Web 2.3483 2.3475 2.3518 2.3539 2.358 2.4361 2.4585 4.3158 Book-ppt 0.0218 0.022 0.0221 0.0222 0.0221 0.0225 0.6049 3.7392 Linux 0.0046 0.0046 0.0047 0.0047 0.0047 0.0046 0.0072 3.5543 Video-image 0.6136 0.6136 0.6136 0.6136 0.6136 0.6611 0.6919 3.6136 5.3 Summary File Aware Deduplication system works at the file level. It is mainly designed to enhance the performance of the deduplication system, where the workloads have weak temporal locality and have domination of small files accesses over large files accesses. Apart from this, large files have a different level of data redundancy. FADD system is considering file semantics such as size and type to categorize files as highly duplicate, low duplicate and unpredictable duplicate files. In order to overcome deduplication metadata access over- head, suitable separate metadata structures are used for each category. To improve read performance, segment level deduplication for high and unpredictable duplicate files and whole file deduplication for low duplicate files is applied. File type-specific deduplication saves resources and reduces deduplication overhead. In the experiments conducted, it is observed that the FADD system performed better than other systems. This work can be extended to implement a distributed deduplication system, consisting of multiple storage nodes and clients. 125

level similarity-based index tables and for L type files index table with file id and whole file hash is maintained.

Local deduplication: For H or U type files, when a superchunk is received for deduplication, segment level deduplication is performed. Whereas, for L type files, whole file deduplication is applied.

6.1.2 Similar segments identification

Segment level deduplication is performed on files enclosed under H and U types. In this context, to identify similar segments, Broder's theorem [80] is found to be very effective.

Suppose $S1$ and $S2$ represent segments and their corresponding set of hashes of data

blocks are given by $H(S1)$ and $H(S2)$. H is chosen uniformly and at random from a min-wise independent family of permutations. Let $min(H(S1))$ and $min(H(S2))$ represents minimum fingerprints of set $S1$ and $S2$ respectively. Broder's theorem states that, if two segments share many blocks then those segments are highly similar. Thus the probability that their minimum fingerprints are the same, is very high and is equivalent to their Jaccard similarity coefficient as given in Equation 6.1.

$$Pr[min(H(S1) = min(H(S1)] = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} \tag{6.1}$$

Fingerprints, for all of the data blocks of each segment, are computed using MD-5 algorithm. Minhash among those computed fingerprints is taken as the Representative IDentifier (RID) of the segment. The set of segments having same RID are considered as a group of similar segments and are mapped to the same bucket. Bucket is a logical store for keeping the information about similar segments and the structure of a bucket is given in the following subsection.

6.1.3 Metadata structures

Metadata stored at the coordinator, data servers and clients, is shown in Figure 6.2.

Coordinator maintains a mapping table for file type and addresses of data servers handling that file type. For each data server, its capacity, used block count and an array of LogLog counters are maintained. LogLog counters are used to measure the cardinality of the data set stored on each server.

Data servers handling H and U type files maintain RID table, buckets of similar segments, LBA-to-PBA and PBA-to-Bucket mapping tables. RID table is used to index the buckets. Bucket consists of the metadata of similar segments. Buckets are maintained as an array of structures, which have members - segments of physical blocks, their fingerprints and reference counts. In order to handle bucket overflow condition, due to the addition of more segments, a link bucket is appended. Thus, chain of buckets is used to store metadata of huge number of similar segments together. RID index, LBA-to-PBA, and PBA-to-Bucket mapping tables are organized as B-trees. For read request processing,

the LBA-to-PBA mapping table is used to locate the required target physical blocks. When a write operation is issued, in the process of deduplication, PBA reference counts need to be modified. In this context PBA-to-Bucket mapping table is required to locate the PBA entry. Data servers handling L type files use whole file hash-based B-tree index and LBA-to-PBA maps.

Client maintains a file recipe table which consists of information such as file id, starting LBA, size in terms of blocks, list of superchunks and the corresponding addresses of the data servers on which the superchunks are stored. For L type files, the whole file is considered as one unit and the list of superchunks is replaced with single entry.

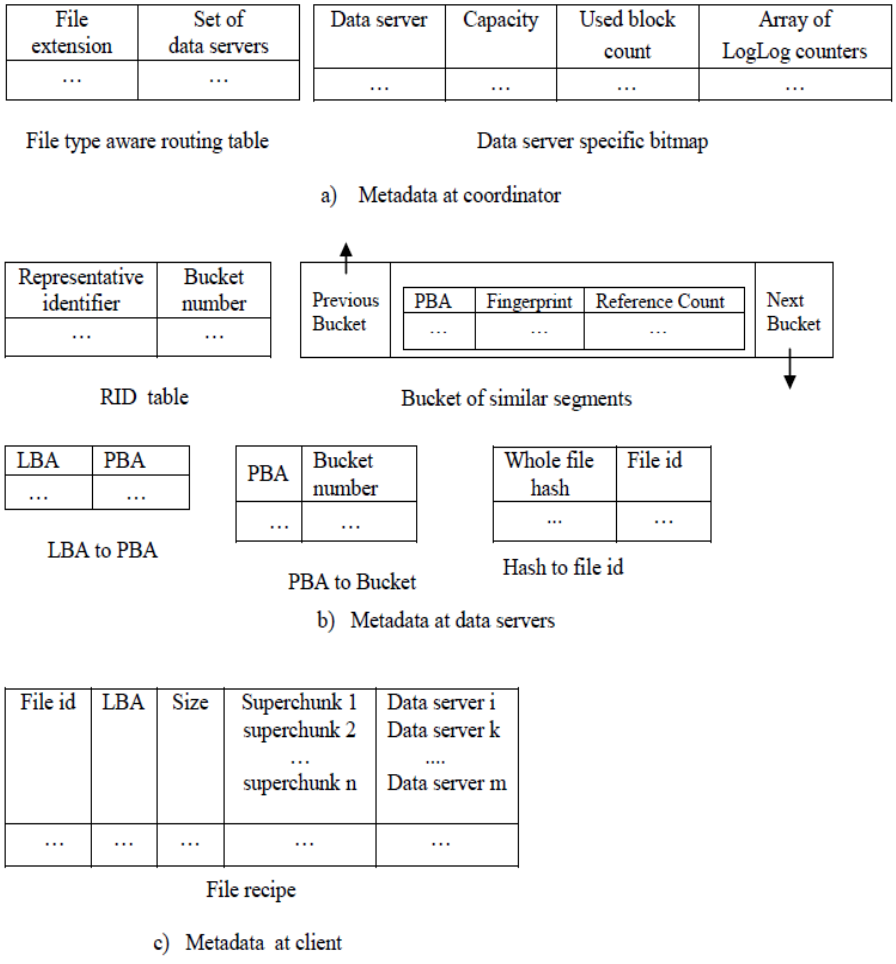


Figure 6.2: Metadata management

6.1.4 Workflow of DDS for L type files

A request for storing a L type file is processed in three steps - preprocessing, data routing and deduplication at data server.

6.1.4.1 Preprocessing

File pre-process module at client extracts extension of the file to determine file category. If extension is not provided, the file type is treated as U type. For L type file, whole file chunking is applied and whole file hash is computed using MD-5 hashing algorithm.

6.1.4.2 Data routing

For the L type files, steps involved for determining data server are shown in Algorithm 6.1. The extension of L type file and whole file hash are sent to the coordinator. Coordinator searches in file type aware routing table to identify set of data servers to handle that file type. The whole file hash is hashed to map to the respective data server. The selected data server performs index lookup for hash. The selected data server sends information about existence of hash to the coordinator. The coordinator sends the address of the selected data server along with the indication of existence of the file, to the client.

6.1.4.3 L type file deduplication at data server

Whole file dedplication is applied on L type files. The detailed steps are shown in Algorithm 6.2. If whole file hash exists that means a copy of the file is already present at the data server, then the corresponding metadata is updated at the data server. In addition to this, a file recipe entry is created at the client. If whole file hash doesn't exist that means the file is not present at the data server. Hence all the data blocks of the file are transferred by the file transfer module and are stored at the data server. Metadata is created for the new file at the data server and a file recipe entry is added at the client.

The interaction diagrams for the file storage process of L type is shown in Figure 6.3. For L type files, no state is remembered and stateless routing is followed based on the whole file hash value.

Algorithm 6.1 Data routing for L type file

Input: File extension and whole file hash

Output : Data server address

```
// Client
Send file extension and whole file hash to the
    coordinator.
// Coordinator
Lookup routing table and identify set of data servers.
Apply hashing on whole file hash to select a data
    server.
// Data server
Selected data server performs index lookup to identify
    the duplicate hash.
Send existence information of whole file hash to coordinator.
// Coordinator
if whole file hash is found then
    Send information about data server address and
        existence of whole file hash on that data server
        to the client.
else
    Determine a suitable data server and send
        data server address and whole file hash
          existence information to client.
end if
```

Algorithm 6.2 L type file deduplication

Input : Data server address and file

1: *// Client*
2: **if** whole file hash exists **then**
3: Update metadata at the data server.
4: **else**
5: *// Client*
6: Send file and store at the data server.
7: *// Data server*
8: Initialize the metadata for the stored file.
9: **end if**
10: *// Client*
11: Add file recipe entry at the client.
12: *// Coordinator*
13: Update used block count of the selected data server, if required.

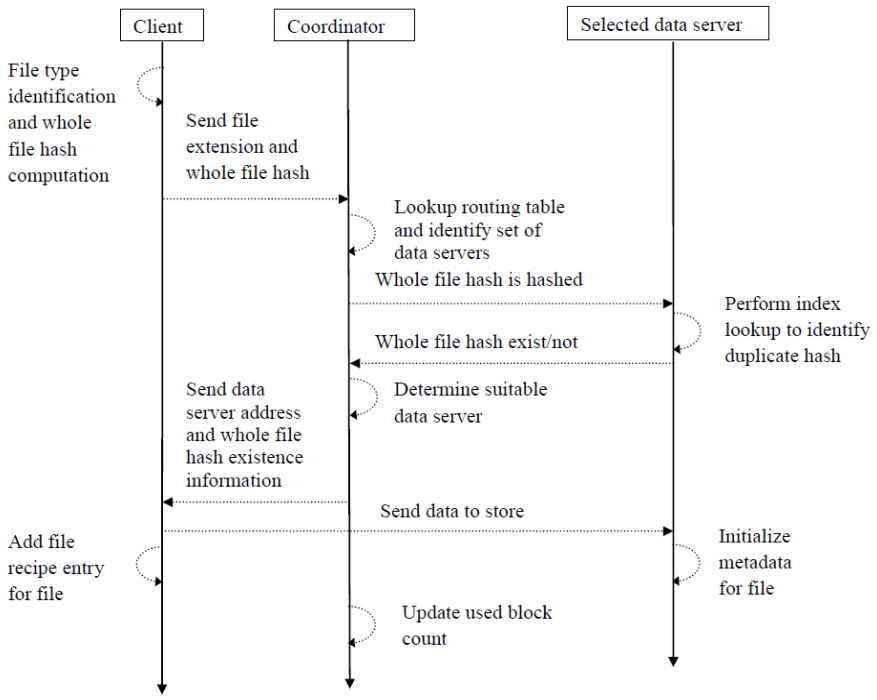


Figure 6.3: Interaction diagram for L type file storage

6.1.5 Workflow of DDS for H and U type files

A request for storing a H and U type file is processed in three steps - preprocessing, data routing and deduplication at data server.

6.1.5.1 Pre-processing

File pre-process module at client extracts extension of the file to determine the file category. If file extension is not given, it is treated as U type file. Based on the category of the file, it creates a list of superchunks for H and U type files using Algorithm 6.3. The file is partitioned into superchunks of size 1 MB. Further, each superchunk is divided into segments of size 64 KB. Each segment is fixed size chunked to generate data blocks of size 4 KB. Hash is computed for each data block using MD-5 hashing algorithm. Minimum hash among all hashes of data blocks of a segment is taken as representative of the segment.

6.1.5.2 Data routing

The client sends file extension and the set of minhashes of a superchunk to the coordinator. The coordinator searches through file type aware routing table to identify set of data servers to handle that file type. The coordinator selects best data server among set of data severs for suprechunk storage, based on duplicate block count and storage usage at each data server. At coordinator, for each data server, the sets of m counters are tentatively updated using the minhashes of the superchunk. Average value(Q) of the m-counters is computed and the cardinality of the set i.e., unique data block count on that server is estimated using Equation 6.2. Weighted duplicate block count is computed using Equation 6.5. The procedure for estimating cardinality and storage usage is given as follow.

Algorithm 6.3 Two level chunking

Input : File $f(LBA$: starting address, $size$: number of blocks, D : sequence of data blocks)

Output : SC: File superchunks

```
Identify file type as H or U type.
Divide the file data into fixed size (= 1 MB)
  superchunks, SC_i, 1 ≤ i ≤ n.
for i = 1 to n do
    Divide the superchunk SC_i into segments (S_j) of
      data (D_ij) of size 64 KB.
    for each segment S_j do
        Set of fingerprints S_j.F ← φ
        Set of data blocks S_j.D ← φ
        Divide the segment data D_ij into fixed size
        (4 KB) data blocks D_1, D_2, ..., D_m.
        for k = 1 to m do
            fp ← MD − 5(D_k)
            S_j.F ← S_j.F ∪ fp
            S_j.D ← S_j.D ∪ D_k
        end for
        S_j.RID ← Minhash(S_j.F)
    end for
end for
```

LogLog counter: LogLog counter [101] is a probabilistic data structure that is used to measure the cardinality of the data set stored on each data server. Coordinator maintains a set of m 8-bit LogLog counters (here m = 8192), using the multiset of hashes corresponding to the superchunks stored on each of the data servers. Every hash (x) of a chunk is treated as an M-bit binary number, which is divided into $C = \log m$, least significant bits and remaining R bits, as shown in Figure 6.4. C-bit value is used to select the counter and position of the least significant one bit in R-bits is the rank of the hash item. The selected counter is updated if the rank of the hash item is greater than the existing counter value. This procedure is applied for each of the minhashes of a superchunk to update the corresponding counters of the data servers. Average value Q of the m-counters is computed and

the cardinality of the set is estimated using Equation (6.2).

$$Cardinality = 0.721205 \times m \times 2^{Q} \tag{6.2}$$

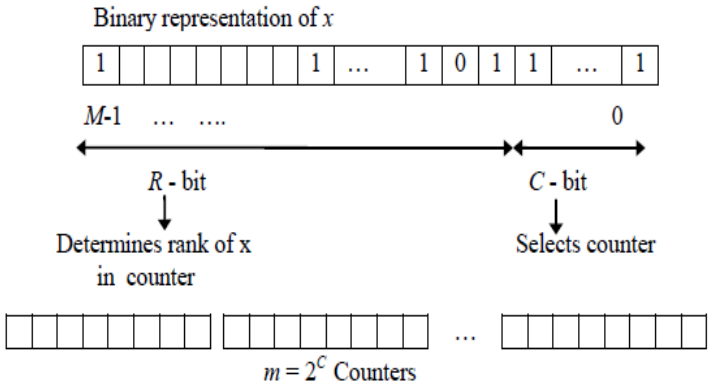


Figure 6.4: LogLog counter

In order to compute duplicate block count of a given super chunk at a data server, let $UC1$ denotes unique block count before updation of LogLog counter, $UC2$ denotes estimated unique block count after updation of LogLog counter using minhashes of the superchunk, $|SC|$ denotes superchunk size, DBC denotes duplicate block count for superchunk (Equation 6.3) and SU storage usage factor (Equation 6.4).

$$DBC = |SC| - (UC2 - UC1) \tag{6.3}$$

$$SU = \frac{Average\ used\ block\ count}{DS_j\ used\ block\ count} \tag{6.4}$$

$$Weighted\ DBC = SU \times DBC \tag{6.5}$$

A data server with maximum weighted duplicate block count is chosen as the target data server for storing a super chunk. The detailed steps for the selection of the data server and the estimation of duplicate block count are given in Algorithms 6.4 and 6.5 respectively.

Algorithm 6.4 Data routing for H and U type

Input : File extension and Superchunk SC_i

Output: Data server address

1: *// Client*
2: Let SC_i^{mf} set of minhashes of superchunk SC_i.
3: Send file extension and SC_i^{mf} to the coordinator.
4: *// Coordinator*
5: Identify the set of data servers DS handling the file type.
6: $maxwdbc \leftarrow DOUBLE\_MIN$
7: **for** each $DS_j \in DS$ **do**
8: $UC1 \leftarrow Cardinality(DS_j.counters)$
9: $tcounters \leftarrow UpdateCounters(DS_j.counters, SC_i^{mf})$
10: $UC2 \leftarrow Cardinality(tcounters)$
11: $SU \leftarrow \frac{Average\ used\ block\ count}{DS_j\ used\ block\ count}$
12: $w \leftarrow SU \times (|SC| - (UC2 - UC1))$
13: **if** $w > maxwdbc$ **then**
14: $maxwdbc \leftarrow w$
15: $sel\_DS \leftarrow DS_j$
16: **end if**
17: **end for**
18: return sel_DS
19: Cardinality(LogLogCounters)
20: $Q \leftarrow average(LogLogCounters)$
21: Return $(0.721205 \times m \times 2^Q)$

6.1.5.3 H and U type file deduplication at data server

For each superchunk of H and U type files, the file transfer module sends hash values of the data blocks to the selected data server. The data server divides this sequence of hashes into segments of 16-hashes. For each 16-hash segment, RID is found and the corresponding bucket is identified, by searching the RID index. If an entry is not found, a new bucket is allocated, metadata of the segment is added to that bucket. If the bucket is already existing,

Algorithm 6.5 Update LogLog counters

Input : OldLogLogCounters $OLLCounters$ and Minhashes of Superchunk SC^{mf}

Output : NewLogLogCounters $NLLCounters$

```
for i = 0 to m-1 do
    NLLCounters[i] ← 0
end for
for each minhash f ∈ SC^mf do
    Let C be the least significant log2m bits and
        R be the remaining bits of f
    rank ← least significant 1-bit position in R
    if OLLCounters[C] < rank then
        NLLCounters[C] ← rank
    else
        NLLCounters[C] ← OLLCounters[C]
    end if
end for
return NLLCounters
```

then the duplicates are identified and the corresponding reference counts are incremented if required. The hashes which are not found in the bucket are added with an initialized reference count of one and the block number is added to the list of missing blocks. This procedure is repeated for all 16-hash segments of the superchunk. The identified missing blocks list is communicated to the file transfer module. The file transfer module sends only the data of the missing blocks, which reduces the load on the network, and the data server stores those received missing data blocks. Both of the client and the data servers update the index tables and the coordinator updates the LogLog counters and used block count corresponding to the selected data server. These steps are repeated by the client for all the superchunks. The detailed steps are shown in Algorithm 6.6. The interaction diagram for the file storage process of H and U type files is shown in Figure 6.5. The state information remembered by the coordinator for determining the data routing for H and U type files is minimal which makes this architecture more scalable.

In order to retrieve a file, the client identifies the file's superchunks list and respective data server addresses from the file recipe, for H and U type files. For L type file, one

data server address is identified from the file recipe. The client sends file/superchunk read requests to the respective data servers. Each data server retrieves file/superchunks stored on it and sends the data to the client. At the client, the file is reconstructed from the received data. Interaction diagram for the file retrieval process is shown in Figure 6.6.

Algorithm 6.6 H and U type file deduplication

Input : Data server address and superchunk

```
1:  // Client
2:  Let F be the fingerprints of all the blocks of
3:    superchunk
4:  Send F to the selected data server
5:  // Data server
6:  Divide F into 16-hash segments.
7:  for each 16-hash segment do
8:      Find RID (Minhash).
9:      Search RID-index, for a matching bucket.
10:     if bucket not found then
11:         Allocate a new bucket.
12:         Add metadata of the segment to the bucket.
13:         Update RID-index.
14:         Add all of the block numbers to the list of
15:           missing blocks.
16:     else
17:         Search the bucket for the fingerprints
18:       of the segment.
19:         if fingerprint present then
20:             Increment reference counts, if required.
21:         else
22:             Add an entry with the new fingerprint.
23:             Initialize the reference count to one
24:             Add the block number to the list of
25:             missing blocks.
26:         end if
27:     end if
28: end for
29: Send the list of missing blocks to the client
```

Algorithm 6.6 continuation

30: *// Client*
31: Client sends the data of the missing blocks
32: (Add file recipe entry for the entire file).
33: *// Data server*
34: Data server stores the received missing blocks.
35: *// Coordinator*
36: Update the LogLog counters and used block count
37: of the selected data server.

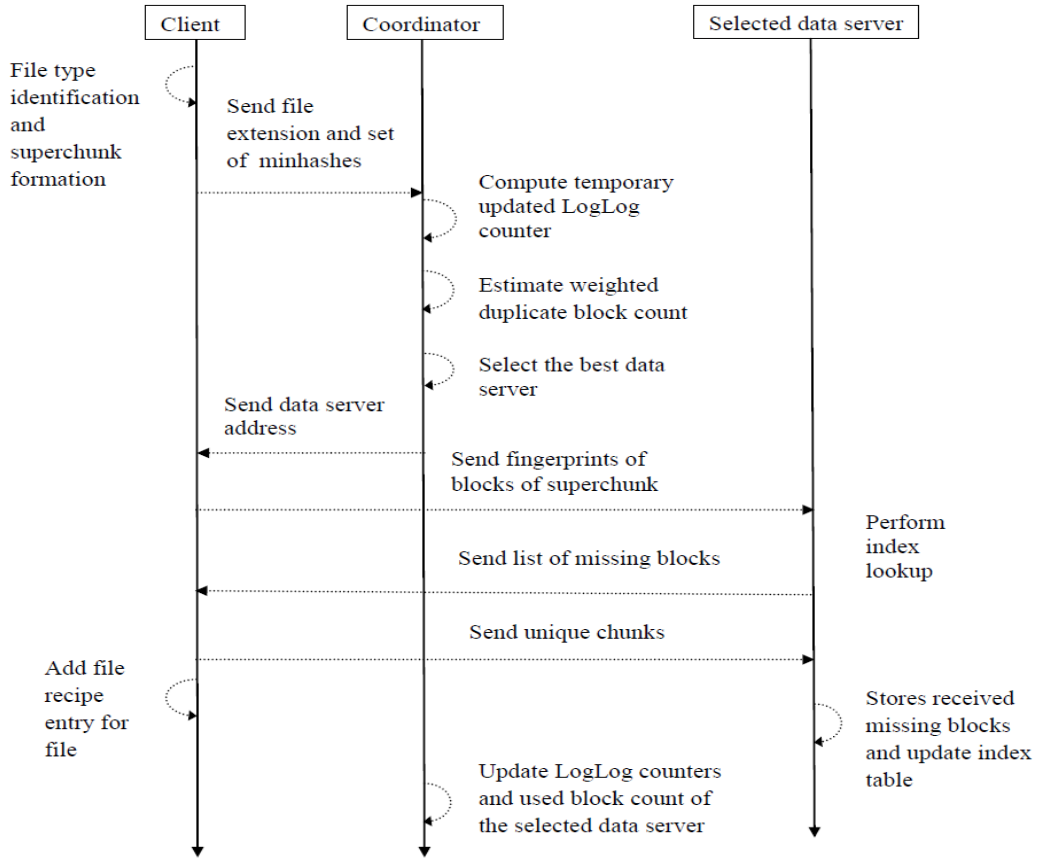


Figure 6.5: Interaction diagram for H and U type file storage

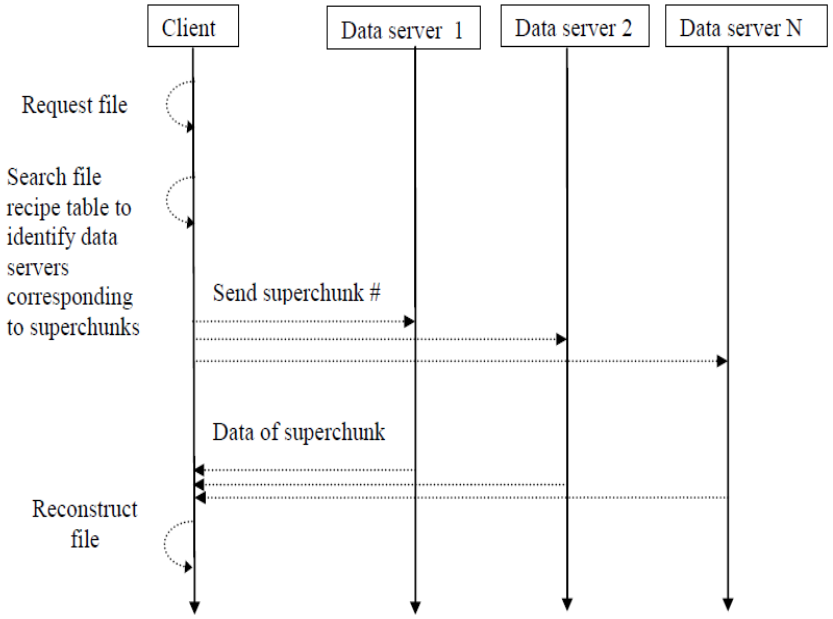


Figure 6.6: Interaction diagram for file retrieval process

6.2 Experimental results

Prototype of the DDS system and extreme binning are implemented and simulated under the Linux operating system running on Intel i7 processor based system. Trace driven experiments are conducted to assess system performance. Traces include standard I/O traces taken from two production systems at FIU and some locally collected data sets. The standard I/O traces consist of the I/O requests generated by the virtual machines running web server (Web) and email server (Mail) [22], for a duration of 21 days. In addition to this, the *Linux* dataset, a collection of Linux kernel source code with version 5.x.y, *Video-image* dataset consisting of audio, video and images, and *Book-ppt* data set consisting of books, documents and ppts. The datasets are categorized as H-type (Book-ppt and Linux), U-type (Mail and Web) and L-type (Video-image). For Mail and Web datasets (traces available without data), a file is identified as a sequence of read/write requests from the same process for the consecutive LBAs. Trace statistics - total I/O requests, read requests, write requests, working set size (KB), percentage of duplicate data, count of files, minimum size of file (KB), maximum size of files (KB) and average size of file (KB) for all the data sets used

are given in Table 6.1.

Table 6.1: Trace statistics

	Mail	Web	Linux	Book-ppt	Video-image
Total requests	460334027	14294158	86418389	4835460	21838832
Read requests	51348252	3116456	205236432	47644784	162206568
Write requests	408985775	11177702	86418389	9745351	55777601
Working set size (KB)	58966824	2196696	345673556	38981404	87355328
% of duplicate data	14.32	19.10	98.84	44.46	41.01
Count of files	855065	176982	22454899	80037	44392
Minimum file size (KB)	4	4	4	4	4
Maximum file size (KB)	37628	4096	16032	2978976	2926528
Average file size (KB)	69	12	15	487	5026

Experiments are conducted by varying the number of data servers from 2 to 64. In the study, the parameters (i) data skew (ii) normalized space saving (iii) read throughput (iv) write throughput and (v) assignment time are measured.

Data skew is defined as the ratio of storage space on the maximum loaded data server over average storage space utilized. Data skew value indicates load distribution, and the more the data skew value more the load imbalance. The data skew values computed for each data set by varying the number of storage nodes are shown in Table 6.2. The results show that better load balancing is achieved with DDS approach than with extreme binning (EB). Extreme binning exhibits more data skewness, leading to storage imbalance, as data server count increases. This effect is due to stateless routing. Whereas, DDS uses a hybrid routing approach, balances the load and achieves reasonable space saving.

Space saving is computed as the difference between the total amount of data to be stored and the actual amount of data stored by the data server(s). Maximum duplicate content that can be eliminated using full deduplication approach is used to normalize the space saving. Space saving for Book-ppt, Video-image, Mail, Web and Linux datasets are shown in Figures 6.7a, 6.7b, 6.7c, 6.7d and 6.7e respectively. It can be observed that the space saving for extreme binning is constant for all datasets, irrespective of the number of data servers. This is due to the mapping of the same set of files to the same bins, irrespective of the count of data servers and a bin is the unit of search space for identifying the duplicate data. Whereas, for DDS, space saving varies with the count of data servers, except for

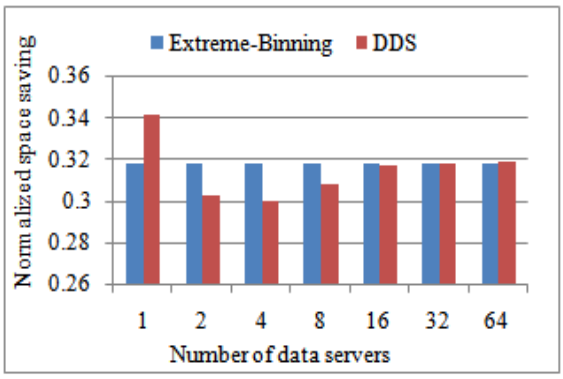
Table 6.2: Data skew values for all datasets

Dataset name	Deduplication Approach	Data skew value					
	No. of data servers →	2	4	8	16	32	64
Book-ppt	DDS	1	1.000062	1.00005	1.000803	1.002406	1.0062971
	EB	1.915654	3.758084	7.450405	14.848305	29.655698	59.277045
Video-image	DDS	1.06437	1.071494	1.19132	1.461122	1.685758	2.111097
	EB	1.986641	3.961469	7.91194	15.815255	31.623849	63.243035
Mail	DDS	1.000002	1	1	1.000848	1.00144	1.008686
	EB	1.485671	2.52853	4.7	9.122751	18.041013	35.928305
Web	DDS	1	1.000142	1.000399	1.000679	1.001339	1.001439
	EB	1.175014	1.513203	2.238789	3.758122	6.868256	7.868256
Linux	DDS	1.000003	1.000005	1	1	1	1.000033
	EB	1.085532	1.257576	1.625398	2.435319	4.20417	7.883083

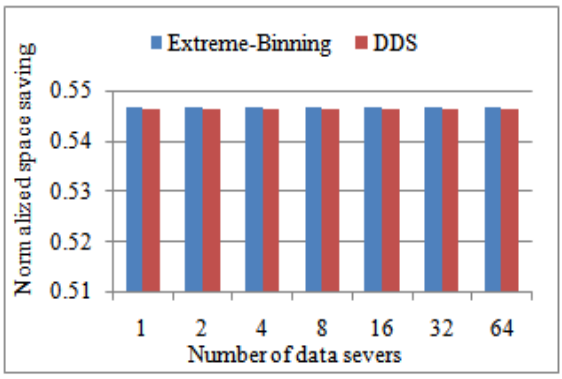
Video-image data set. In the case Video-image dataset whole file mapping approach similar to that of extreme binning is followed, which gives identical space saving value. For other datasets depending on the size of the files and dynamic mapping, the amount of duplicate content identified may vary with the number of data servers.

Read throughput for Book-ppt, Video-image, Mail, Web and Linux datasets are shown in Figures 6.8a, 6.8b, 6.8c, 6.8d and 6.8e respectively. Throughput is measured relatively. Read throughput is consistently increasing along with the increase in the number of data servers for all datasets in DDS compared to extreme binning. DDS performs parallel access of files (different segments from different data servers) during a read operation and extreme binning has to access the whole file from a single data server.

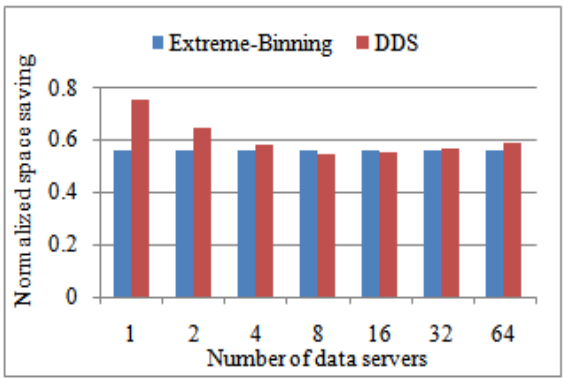
Write throughput for Book-ppt, Video-image, Mail, Web and Linux datasets are shown in Figures 6.9a, 6.9b, 6.9c, 6.9d and 6.9e respectively. Write throughput is consistently increasing along with the increase in the number of data servers for Book-ppt and Video-image dataset. However, for Mail, Web and Linux datasets, write throughput is increasing as number of data servers reaches 8. After this, if more data servers are added, there is decrease in write throughput compared to extreme binning. This is due to the domination of small files in these datasets. Though processing time is reduced/divided for these datasets, assignment overhead is increasing. DDS performs parallel storage of files during the write operation and extreme binning has to store the whole file to a single data server. In the experiments, read and write throughput are measured with respect to the client and one client is used to generate/play the read/write requests using the I/O trace. When processing write requests, the client need not wait for the completion of deduplication of a file/superchunk



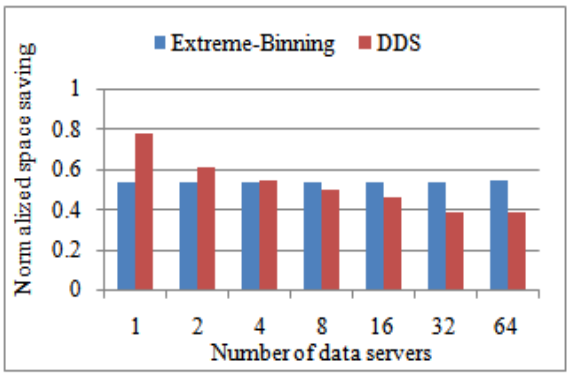
(a) Book-ppt dataset



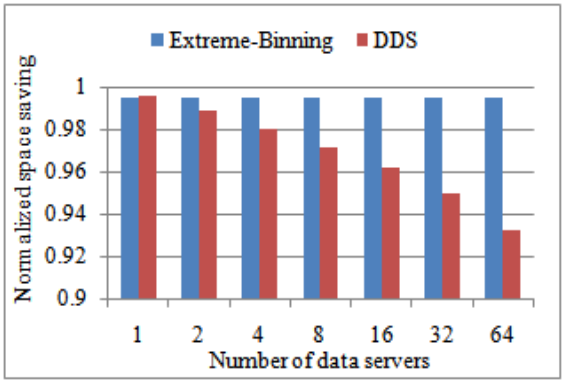
(b) Video-image dataset



(c) Mail dataset

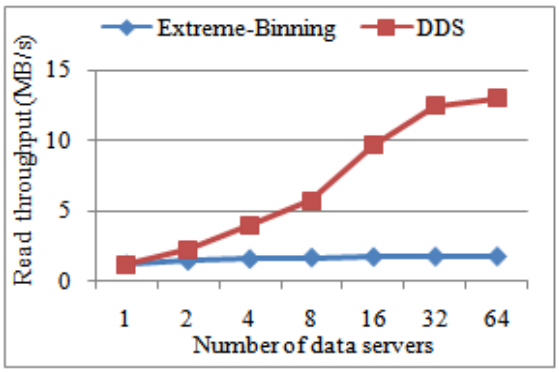


(d) Web dataset

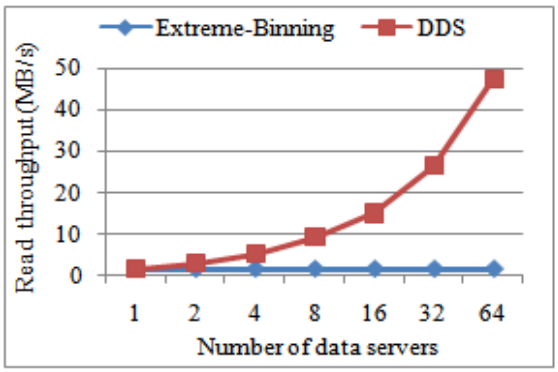


(e) Linux dataset

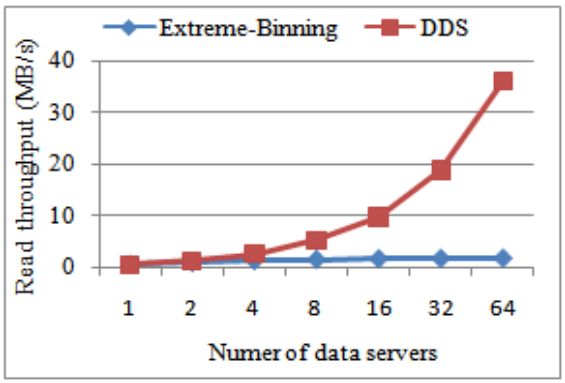
Figure 6.7: Space saving



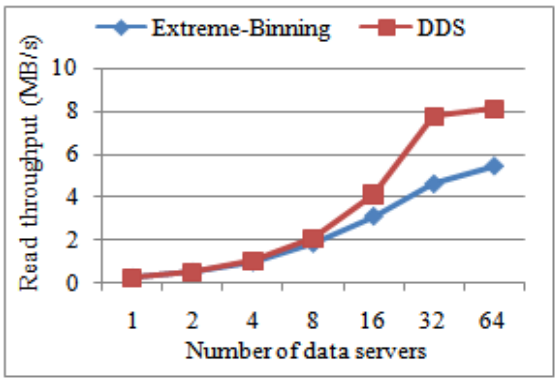
(a) Book-ppt dataset



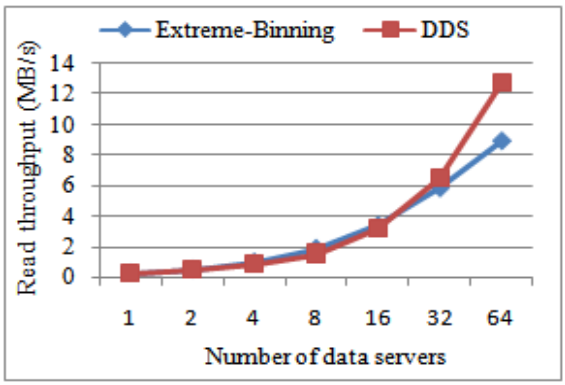
(b) Video-image dataset



(c) Mail dataset

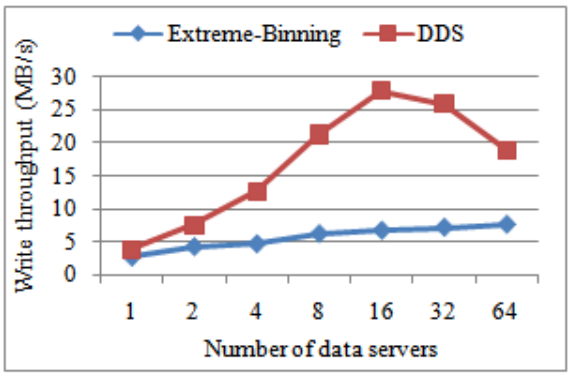


(d) Web dataset

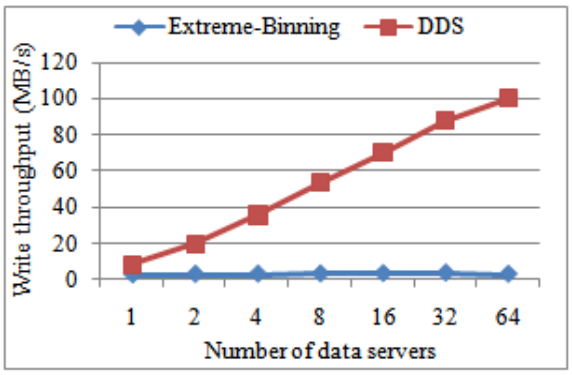


(e) Linux dataset

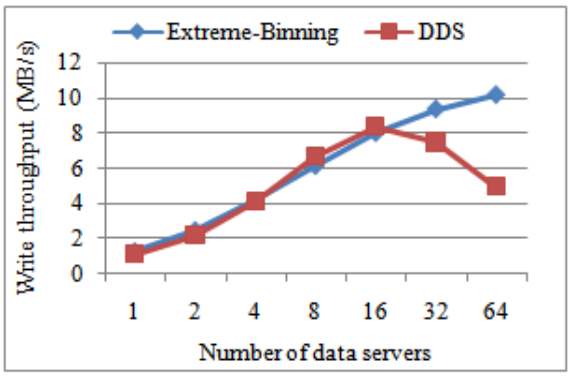
Figure 6.8: Read throughput



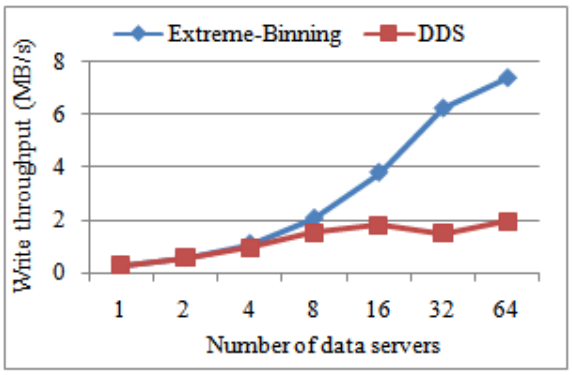
(a) Book-ppt dataset



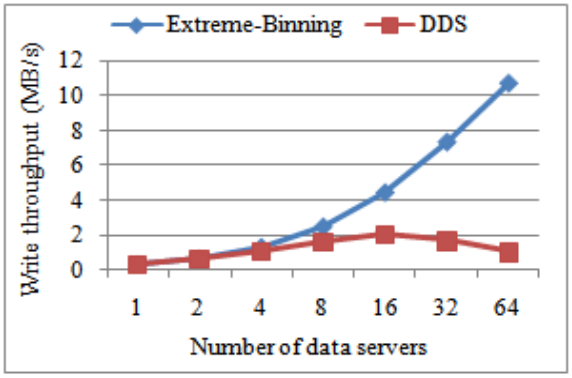
(b) Video-image dataset



(c) Mail dataset



(d) Web dataset



(e) Linux dataset

Figure 6.9: Write throughput

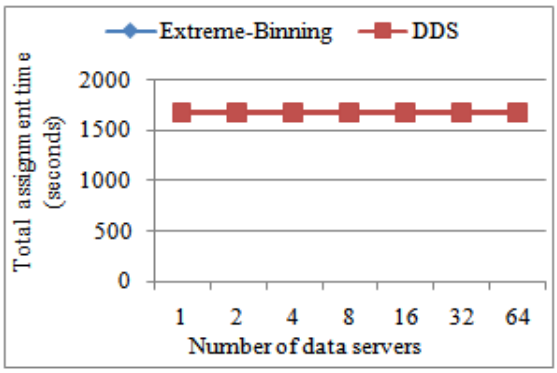
at the data server(s). Whereas, for read request processing, the client waits for the response from the data server(s), before issuing the next read request. So the write throughput can be observed to be higher than the read throughput. Both of the read and write throughputs are measured relatively and do not represent the performance of any particular real storage system.

Given a file, the amount of time required to determine the target data server(s) is called file assignment time. We have measured this parameter for all of the files of each dataset. Assignment time for Book-ppt, Video-image, Mail, Web and Linux datasets are shown in Figures 6.10a, 6.10b, 6.10c, 6.10d and 6.10e respectively. Assignment time increases linearly with data server count for all datasets in DDS except for Video-image dataset. For Video-image dataset the assignment process in DDS is stateless and the time taken is nearly same as extreme binning system.

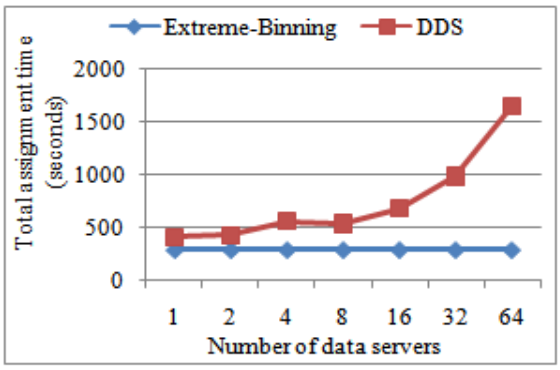
A distributed deduplication system preprocesses the data (chunking and hashing), determines the data server and the (unique) data to be transferred (decision making), and finally transfers the data. Average time required in seconds per 1 MB for preprocessing, decision making and transferring the data is given in table 6.3. Preprocessing time for DDS and extreme binning is equal. Decision making time gives time incurred for inquiring suitable data server and duplicate identification. Decision making time for DDS is more than extreme binning. This is due to the stateful routing approach. Data transfer time gives the time required for transferring data after deduplication. It can be observed that the data transfer time for DDS is less compared to that of extreme binning, because DDS transfers unique data blocks. If the file consists of multiple segments, those segments may be transferred to different data servers in an overlapped manner. Due to this, there may be a slight reduction in the average time for data transfer.

6.3 Summary

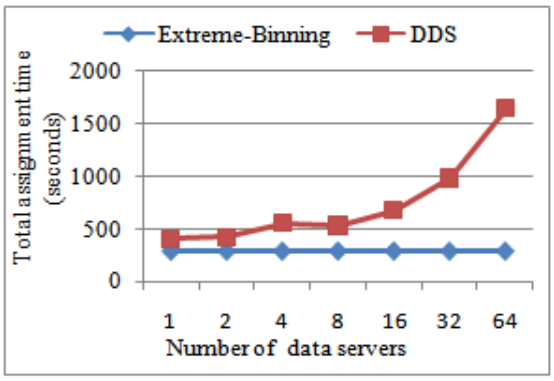
Distributed Deduplication System (DDS) is performing deduplication of files based on the type of file. The files to be deduplicated are broadly classified based on data redundancy into three categories as high, low and unpredictable duplicate content files. Separate data servers for handling each file category are allocated which results in data similarity-based data partitioning. Two distinct approaches are followed for distributing files among data servers. Low duplicate files are distributed using stateless routing. Whereas, high and unpredictable files are distributed using stateful routing. In order to maintain the data locality, file level deduplication for low duplicate files and superchunk level deduplication



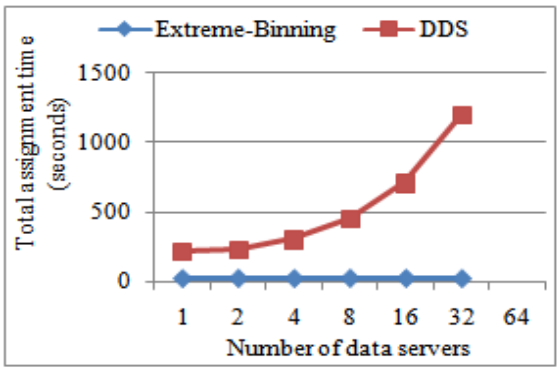
(a) Book-ppt dataset



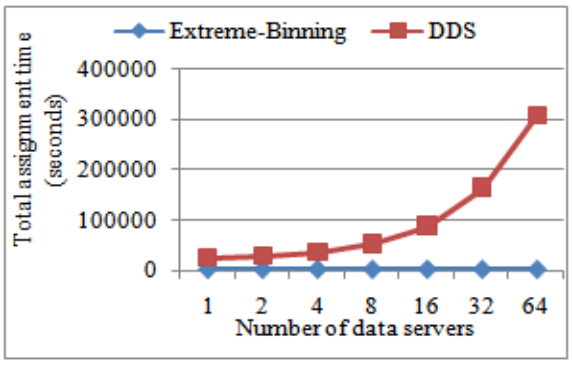
(b) Video-image dataset



(c) Mail dataset



(d) Web dataset



(e) Linux dataset

Figure 6.10: Assignment time

Table 6.3: Time for preprocessing, decision making and data transfer

Dataset name	Time component	Deduplication approach	Average time in milliseconds per 1 MB						
		No. of data servers →	1	2	4	8	16	32	64
Book-ppt	Preprocessing	DDS	7.7	7.7	7.7	7.7	7.7	7.7	7.7
		EB	7.7	7.7	7.7	7.7	7.7	7.7	7.7
	Decision making	DDS	0.9	0.9	1	1.5	2.1	4.3	4.8
		EB	0.6	0.6	0.6	0.6	0.6	0.6	0.6
	Data trasfer	DDS	71.3	36.4	18.2	9.1	4.5	2.26	1.1
		EB	84	80	78.2	77.4	77.1	77.6	76.9
Linux	Preprocessing	DDS	7.7	7.7	7.7	7.7	7.7	7.7	7.7
		EB	7.7	7.7	7.7	7.7	7.7	7.7	7.7
	Decision making	DDS	9.8	10	12	23	32	41	48
		EB	8	8	8	8	8	8	8
	Data trasfer	DDS	1.4	1	0.7	0.4	0.3	0.2	0.1
		EB	87	48.4	29.2	20.1	15.8	13.9	13
Mail	Preprocessing	DDS	0.3	0.3	0.3	0.3	0.3	0.3	0.3
		EB	0.3	0.3	0.3	0.3	0.3	0.3	0.3
	Decision making	DDS	0.1	0.1	0.3	0.9	0.98	1	1.6
		EB	0.1	0.1	0.1	0.1	0.1	0.1	0.1
	Data trasfer	DDS	2.7	1.4	0.7	0.4	0.2	0.1	1
		EB	3	2.2	1.9	1.8	1.7	1.7	1.7
Web	Preprocessing	DDS	0.4	0.4	0.4	0.4	0.4	0.4	0.4
		EB	0.4	0.4	0.4	0.4	0.4	0.4	0.4
	Decision making	DDS	1	1.6	2.8	3.9	5.8	8.9	9.9
		EB	0.5	0.5	0.5	0.5	0.5	0.5	0.5
	Data trasfer	DDS	3.6	1.9	1	0.5	0.2	0.1	0.9
		EB	4.3	2.5	1.7	1.3	1.1	1	1.2
Video-images	Preprocessing	DDS	7.7	7.7	7.7	7.7	7.7	7.7	7.7
		EB	7.7	7.7	7.7	7.7	7.7	7.7	7.7
	Decision making	DDS	0.7	1.4	2.8	3.9	4.9	6.1	7.7
		EB	0.3	0.3	0.3	0.3	0.3	0.3	0.3
	Data transfer	DDS	65.1	34.6	17.4	9.7	5.9	3.4	2.1
		EB	83.9	83.4	83.2	83.1	83.1	83.1	83.1

for duplicate and unpredictable type is applied. By applying a different suitable deduplication approach for each category, a reduction in deduplication overhead is achieved. Apart from this, to achieve a high deduplication ratio and balancing storage space across all data servers, data server with high data similarity is selected probabilistically. Performance of DDS is found to be consistently better in achieving the load balancing with reasonable space saving. In future, this work can be extended to include adaptive fault tolerance mechanism, to improve the system reliability and availability.

Chapter 7

Conclusion and future scope

7.1 Conclusions

This thesis investigates primary storage deduplication schemes for cloud storage. In cloud environment, workloads exhibit random access pattern, weak temporal locality and have domination of small file accesses over large file accesses. Apart from this, large files have a different level of data redundancy. Primary storage deduplication system can be implemented at block level or file level. Irrespective of level, challenges associated with primary storage deduplication system are disk-bottleneck and data fragmentation problems. Disk-bottleneck problem arise due to random access pattern of deduplication metadata. In order to overcome disk-bottleneck problem, similarity based bucket indexing approach is followed. Bucket represents logical container for similar segments. Selective deduplication is used to overcome data fragmentation problem. In addition to this, caching mechanism optimization and file semantic aware deduplication schemes are proposed to reduce deduplication overhead.

In chapter 3 HDS, a block-based hybrid primary storage deduplication system that applies deduplication in the background, is described. HDS uses a similarity-based indexing mechanism to locate all metadata of similar segments in one bucket and reduces the disk-bottleneck problem. Additionally, this type of indexing reduces search space for the identification of duplicates and because of the reduction in the overhead, HDS can be used for primary storage deduplication. HDS preserves the locality order of fingerprints in a bucket

and selective deduplication helps in reducing the data fragmentation. In turn, overall I/O system performance can be improved.

In chapter 4, Hybrid deduplication system with a content-based cache is proposed which works at the block layer. To accelerate deduplication, a prototype of content- based data cache with Modified-ARC is implemented to populate the cache with unique data blocks. For determining the popularity of data blocks, Modified-ARC considers weighted frequency and idle staying period, in addition to the recency of references. It is found that Modified-ARC outperforms LRU and ARC.

In chapter 5, the File Aware Deduplication system which works at the file level is proposed. File semantics such as size and type are used by the FADD system to classify files as high duplicate, low duplicate, or unpredictable duplicate files. To reduce deduplication metadata access overhead, appropriate distinct metadata structures for each category are employed. Segment level deduplication for high and unpredictable duplicate files and whole file deduplication for low duplicate files are used to increase the performance. Deduplication based on file type saves resources and reduces deduplication overhead. During the experiments, it was discovered that the FADD system outperformed the other systems.

In chapter 6, Distributed Deduplication System (DDS) is presented which performs deduplication of files based on their type. The files to be deduplicated are divided into three categories depending on data redundancy: high, low, and uncertain duplicate content files. Separate sets of data servers for handling each file category are allocated which results in similarity-based data partitioning. Two distinct approaches are followed for distributing files among data servers. Low duplicate files are distributed using stateless routing. Whereas, duplicate and unpredictable files are distributed using stateful routing. File level deduplication for low duplicate files and superchunk level deduplication for high duplicate and unpredictable type files are used. Deduplication overhead is reduced by using a separate appropriate deduplication strategy for each file category. Apart from that, data servers with high data similarity are determined probabilistically to achieve a high deduplication ratio and at the same time load balance is achieved among the data servers. Performance of DDS is found to be consistently better in achieving the load balance with reasonable space saving.

All the schemes proposed in chapter 3, 4, 5 and 6 achieves better performance in terms of response time and storage optimization. The proposed methods are experimented on using publicly available standard FIU datasets and some locally collected datasets, and a comparative study of the proposed methods has been presented and demonstrated their merits and capabilities.

7.2 Future scope

- Machine learning based techniques to learn the data redundancy level of files for categorization.
- In distributed deduplication, applying learning automata based replication strategy for data reliability.
- Machine learning based techniques for working set identification and adaptive cache sizing.
- Modified-ARC algorithm for SSD devices to enhance SSD lifetime.

Bibliography

[1] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 7:1–16, 2012.

[2] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup : Device Mapper Target for Data Deduplication. In *2014 Ottawa Linux Symposium*, Ottawa, 2014. Citeseer.

[3] Hongliang Yu, Xu Zhang, Wei Huang, and Weimin Zheng. PDFS: Partially Dedupped File System for Primary Workloads. *IEEE Transactions on Parallel and Distributed Systems*, 28(3):863–876, 2016.

[4] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. A study on data deduplication in HPC storage systems. In *IEEE Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11, Dallas Texas, 2012. IEEE.

[5] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary Data Deduplication – Large Scale Study and System Design. In *Presented as part of the 2012 USENIX Annual Technical Conference USENIX ATC 12*, pages 285–296, Boston, MA, 2012. USENIX.

[6] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7:1–20, 2012.

[7] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *USENIX annual technical conference*, pages 26–30, Portland, OR, USA, 2011. USENIX.

[8] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, volume 12, pages 1–14, San Jose, CA, 2012. USENIX.

[9] Avani Wildani, Ethan L Miller, and Ohad Rodeh. HANDS: A heuristically arranged non-backup in-line deduplication system. In *IEEE 29th International Conference on Data Engineering (ICDE)*, pages 446–457, Brisbane, Australia, 2013. IEEE.

[10] Bin Lin, Shanshan Li, Xiangke Liao, Jing Zhang, and Xiaodong Liu. Leach: an automatic learning cache for inline primary deduplication system. *Springer Frontiers of Computer Science*, 8:175–183, 2014.

[11] Huijun Wu, Chen Wang, Yinjin Fu, Sherif Sakr, Kai Lu, and Liming Zhu. A Differentiated Caching Mechanism to Enable Primary Storage Deduplication in Clouds. *IEEE Transactions on Parallel and Distributed Systems*, 29:1202–1216, 2018.

[12] Biplob K Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In *USENIX annual technical conference*, pages 1–16, Boston, MA, 2010. USENIX.

[13] Xian Chen, Wenzhi Chen, Zhongyong Lu, Peng Long, Shuiqiao Yang, and Zonghui Wang. A Duplication-Aware SSD-Based Cache Architecture for Primary Storage in Virtualization Environment. *IEEE Systems journal*, 11(4):2578–2589, 2015.

[14] Bo Mao, Hong Jiang, Suzhen Wu, and Lei Tian. Leveraging Data Deduplication to Improve the Performance of Primary Storage Systems in the Cloud. *IEEE transactions on computers*, 65(6):1775–1788, 2015.

[15] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9, London, 2009. IEEE.

[16] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST*, volume 8, pages 1–14, San Jose, CA, 2008. FAST.

[17] Jianwei Yin, Yan Tang, Shuiguang Deng, Ying Li, and Albert Y Zomaya. D^3 : A Dynamic Dual-Phase Deduplication Framework for Distributed Primary Storage. *IEEE Transactions on Computers*, 67(2):193–207, 2017.

[18] Yujuan Tan, Hong Jiang, Dan Feng, Lei Tian, Zhichao Yan, and Guohui Zhou. SAM: A Semantic-Aware Multi-tiered Source De-duplication Framework for Cloud Backup. In *2010 39th International Conference on Parallel Processing*, pages 614–623, San Diego, CA, USA, 2010. IEEE.

[19] Yan Tang, Jianwei Yin, and Wei Lo. SAUD: Semantics-Aware and Utility-Driven Deduplication Framework for Primary Storage. In *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 190–197, New York, NY, USA, 2015. IEEE.

[20] Yinjin Fu, Hong Jiang, Nong Xiao, Lei Tian, and Fang Liu. AA-Dedupe: An Application-Aware Source Deduplication Approach for Cloud Backup Services in the Personal Computing Environment. In *2011 IEEE International Conference on Cluster Computing*, pages 112–120, Austin, Texas USA, 2011. IEEE.

[21] Yinjin Fu, Nong Xiao, Hong Jiang, Guyu Hu, and Weiwei Chen. Application-Aware Big Data Deduplication in Cloud Environment. *IEEE transactions on cloud computing*, 7(4):921 – 934, 2017.

[22] Fiu traces web-link. http://iotta.snia.org/traces/390/, 2010.

[23] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of Backup Workloads in Production Systems. In *FAST*, volume 12, pages 1–16, San Jose, CA, USA, 2012. USENIX.

[24] Yoshihiro Tsuchiya and Takashi Watanabe. DBLK: Deduplication for primary block storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5, Denver, CO, 2011. IEEE.

[25] Sean Quinlan and Sean Dorward. Venti: A New Approach to Archival Storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, volume 2, pages 89–101, California, Santa Cruz, 2002. FAST.

[26] Jingxin Feng and Jiri Schindler. A deduplication study for host-side caches in virtualized data center environments. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, CA, USA, 2013. IEEE.

[27] Debadatta Mishra, Purushottam Kulkarni, and Raju Rangaswami. Synergy: A Hypervisor Managed Holistic Caching System. *IEEE Transactions on Cloud Computing*, 7(3):878–892, 2017.

[28] Jianzhe Tai, Deng Liu, Zhengyu Yang, Xiaoyun Zhu, Jack Lo, and Ningfang Mi. Improving Flash Resource Utilization at Minimal Management Cost in Virtualized Flash-Based Storage Systems. *IEEE Transactions on Cloud Computing*, 5(3):537–549, 2015.

[29] Zhuan Chen and Kai Shen. OrderMergeDedup : Efficient, Failure-Consistent Deduplication on Flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, Santa Clara, CA, 2016. uSENIX.

[30] Ricardo Koller and Raju Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance . *ACM Transactions on Storage (TOS)*, 6(3):13, 2010.

[31] Bin Lin, Shanshan Li, Xiangke Liao, Xiaodong Liu, Jing Zhang, and Zhouyang Jia. CareDedup: Cache-Aware Deduplication for Reading Performance Optimization in Primary Storage. In *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, pages 1–10, Changsha, China, 2016. IEEE.

[32] Bin Lin, Shanshan Li, Xiangke Liao, and Jing Zhang. Rededup: Data Reallocation for Reading Performance Optimization in Deduplication System. In *2013 International Conference on Advanced Cloud and Big Data*, pages 117–124, Nanjing, China, 2013. IEEE.

[33] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Read-Performance Optimization for Deduplication-Based Storage Systems in the Cloud. *ACM Transactions on Storage (TOS)*, 10(2):1–22, 2014.

[34] Ryan NS Widodo, Hyotaek Lim, and Mohammed Atiquzzaman. A new content-defined chunking algorithm for data deduplication in cloud storage. *Future Generation Computer Systems*, 71:145–156, 2017.

[35] A Venish and K Siva Sankar. Study of Chunking Algorithm in Data Deduplication. In *Springer Proceedings of the International Conference on Soft Computing Systems*, pages 13–20, Chennai, India, 2016. Springer.

[36] Yucheng Zhang, Dan Feng, Hong Jiang, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. A Fast Asymmetric Extremum Content Defined Chunking Algorithm for Data Deduplication in Backup Storage Systems. *IEEE Transactions on Computers*, 66(2):199–211, 2016.

[37] Erik Kruus, Cristian Ungureanu, and Cezary Dubnicki. Bimodal Content Defined Chunking for Backup Streams. In *8th USENIX Conference on File and Storage Technologies*, pages 239–252, San Jose, CA, 2010. USENIX.

[38] Calicrates Policroniades and Ian Pratt. Alternatives for Detecting Redundancy in Storage Systems Data. In *USENIX Annual Technical Conference, General Track*, pages 73–86, Boston, MA, 2004. IEEE.

[39] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. P-Dedupe: Exploiting Parallelism in Data Deduplication System. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, pages 338–347. IEEE, 2012.

[40] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Performance Evaluation*, 79:258–272, 2014.

[41] Bhavish Agarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. EndRE: An End-System Redundancy Elimination Service for Enterprises. In *2010 Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI*, pages 419–432, San Jose, CA, USA, 2010. USENIX.

[42] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. AE: An Asymmetric Extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345, Hong Kong, 2015. IEEE.

[43] Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. *IETF RFC 1951*, 1996.

[44] Mark R Nelson. Lzw data compression. *Dr. Dobb's Journal*, 14(10):29–36, 1989.

[45] MFXJ Oberhumer. Lzo-a real-time data compression library. *http://www. oberhumer. com/opensource/lzo/*, 2008.

[46] Val Henson. An Analysis of Compare-by-hash. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue Hawaii, 2003. USENIX.

[47] John Black. Compare-by-Hash: A Reasoned Analysis. In *USENIX Annual Technical Conference, General Track*, pages 85–90, Boston MA, 2006. USENIX.

[48] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A Low-bandwidth Network File System. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, Banff Alberta Canada, 2001. IEEE.

[49] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Fast*, volume 9, pages 111–123, San Francisco, CA, USA, 2009. USENIX IEEE.

[50] Navendu Jain, Michael Dahlin, and Renu Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Fast*, volume 5, pages 21–21, CA, USA, 2005. USENIX.

[51] Girum Dagnaw, Wang Hua, and Ke Zhou. CACH-Dedup: Content Aware Clustered and Hierarchical Deduplication. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 399–407, Sentosa, Singapore, 2018. IEEE.

[52] Yushi Sun, Catherine Y Zeng, Jaeyoon Chung, and Zhe Huang. Online data deduplication for in-memory big-data analytic systems. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7, Paris, 2017. IEEE.

[53] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. Hydrastor: A scalable secondary storage. In *7th USENIX Conference on File and Storage Technologies(FAST '09)*, volume 9, pages 197–210, San Francisco, CA, USA, 2009. USENIX IEEE.

[54] Waraporn Leesakul, Paul Townend, and Jie Xu. Dynamic Data Deduplication in Cloud Storage. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pages 320–325, Oxford, United Kingdom, 2014. IEEE.

[55] Xiaolong Xu and Qun Tu. Data Deduplication Mechanism for Cloud Storage Systems. In *2015 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pages 286–294, Xi'an, China, 2015. IEEE.

[56] Yongtao Zhou, Yuhui Deng, Laurence T Yang, Ru Yang, and Lei Si. LDFS: A Low Latency In-Line Data Deduplication File System. *IEEE access*, 6:15743–15753, 2018.

[57] Charles B Morrey III and Dirk Grunwald. Content-Based Block Caching. In *23rd IEEE, 14th NASA Goddard Conference on Mass Storage Systems and Technologies, MSST*, volume 6, College Park, MD, USA, 2006. Citeseer.

[58] João Paulo and José Pereira. Efficient Deduplication in a Distributed Primary Storage Infrastructure. *ACM Transactions on Storage (TOS)*, 12(4):1–35, 2016.

[59] Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, and Yaping Wan. DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12, Atlanta, GA, 2010. IEEE.

[60] Jibin Wang, Zhigang Zhao, Zhaogang Xu, Hu Zhang, Liang Li, and Ying Guo. I-sieve: An inline high performance deduplication system used in cloud storage. *Tsinghua Science and Technology*, 20:17–27, 2015.

[61] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14, Incline Village, Nevada, USA, 2010. IEEE.

[62] Panfeng Zhang, Ping Huang, Xubin He, Hua Wang, and Ke Zhou. Resemblance and mergence based indexing for high performance data deduplication. *Journal of Systems and Software*, 128:11–24, 2017.

[63] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pages 183–197, San Jose CA, 2013. USENIX Association.

[64] Wei Dong, Fred Douglis, Kai Li, R Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in Scalable Data Routing for Deduplication Clusters. In *10th USENIX Conference on File and Storage Technologies*, volume 11, pages 15–29, San Jose, CA, 2011. USENIX IEEE.

[65] Xin Du, Weizheng Hu, Qiang Wang, and Fang Wang. ProSy: A similarity based inline deduplication system for primary storage. In *IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 195–204, Riverside, CA, USA, 2015. IEEE.

[66] Yujuan Tan, Baiping Wang, Jian Wen, Zhichao Yan, Hong Jiang, and Witawas Srisaan. Improving resImproving Restore Performance in Deduplication-Based Backup Systems via a Fine-Grained Defragmentation Approach. *IEEE Transactions on Parallel and Distributed Systems*, 29(10):2254–2267, 2018.

[67] Shengmei Luo, Guangyan Zhang, Chengwen Wu, Samee Khan, and Keqin Li. Boafft: Distributed Deduplication for Big Data Storage in the Cloud. *IEEE transactions on cloud computing*, 8(4), 2015.

[68] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing Fragmentation for In-line Deduplication Backup Storage via Exploiting Backup History and Cache Knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2015.

[69] Yinjin Fu, Hong Jiang, and Nong Xiao. A Scalable Inline Cluster Deduplication Framework for Big Data Protection. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 354–373, Beijing, China, 2012. Springer.

[70] Jian Liu, Yunpeng Chai, Chang Yan, and Xin Wang. A Delayed Container Organization Approach to Improve Restore Speed for Deduplication Systems. *IEEE transactions on parallel and distributed systems*, 27(9):2477–2491, 2015.

[71] David Frey and Anne-Marie Kermarrec. Probabilistic deduplication for cluster-based storage systems. *Proc. ACM Cloud Computing*, (7):1–14, 2012.

[72] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. Improving Restore Performance in Deduplication Systems via a Cost-Efficient Rewriting Scheme. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):119–132, 2018.

[73] Bo Hong, Demyn Plantenberg, Darrell DE Long, and Miriam Sivan-Zimet. Duplicate Data Elimination in a SAN File System. In *In Proceedings of the 21st IEEE / 12th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST)*, pages 301–314, College Park, MA,USA, 2004. IEEE.

[74] Dirk Meister and André Brinkmann. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6, Incline Village, Nevada, USA, 2010. IEEE.

[75] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Similarity and Locality Based Indexing for High Performance Data Deduplication. *IEEE transactions on computers*, 64(4):1162–1176, 2014.

[76] Suzhen Wu, Xiao Chen, and Bo Mao. Exploiting the Data Redundancy Locality to Improve the Performance of Deduplication-Based Storage Systems. In *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 527–534, Wuhan, China, 2016. IEEE.

[77] Young Jin Nam, Dongchul Park, and David HC Du. Assuring Demanded Read Performance of Data Deduplication Storage with Backup Datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 201–208, Washington, DC, 2012. IEEE.

[78] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, Philadelpia, PA, 2014. USENIX.

[79] Kave Eshghi and Hsiu Khuern Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. *Hewlett-Packard Labs Technical Report TR*, 30:1–11.

[80] Andrei Z Broder. On the resemblance and containment of documents. In *IEEE Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29, Salerno, Italy, 1997. IEEE.

[81] Jurgen Kaiser, Dirk Meister, André Brinkmann, and Sascha Effert. Design of an exact data deduplication cluster. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, San Diego, CA, 2012. IEEE.

[82] Rongyu Lai, Yu Hua, Dan Feng, Wen Xia, Min Fu, and Yifan Yang. A near-exact defragmentation scheme to improve restore performance for cloud backup systems. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 457–471, Xiamen, China, 2014. Springer.

[83] Zhihao Huang, Hui Li, Xin Li, and Wei He. SS-dedup: A high throughput stateful data routing algorithm for cluster deduplication system. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 2991–2995, Washington D.C., USA., 2016. IEEE.

[84] Wenbin Yao, Mengyao Hao, Yingying Hou, and Xiaoyong Li. FASR: An Efficient Feature-Aware Deduplication Method in Distributed Storage Systems. *IEEE Access*, 10:15311–15321, 2022.

[85] PM Ashok Kumar, E Pugazhendhi, and K Vara Lakshmi. Cloud Data Storage Optimization by Using Novel De-Duplication Technique. In *2022 4th International Conference on Smart Systems and Inventive Technology (ICSSIT)*, pages 436–442, Tirunelveli, Tamil Nadu, India, 2022. IEEE.

[86] Geyao Cheng, Deke Guo, Lailong Luo, Junxu Xia, and Siyuan Gu. LOFS: A Lightweight Online File Storage Strategy for Effective Data Deduplication at Network Edge. *IEEE Transactions on Parallel & Distributed Systems*, 33(10):2263–2276, 2022.

[87] Suzhen Wu, Chunfeng Du, Weidong Zhu, Jindong Zhou, Hong Jiang, and Bo Mao. EaD: ECC-assisted Deduplication with High Performance and Low Memory Overhead for Ultra-Low Latency Flash Storage. *IEEE Transactions on Computers*, 10:15311–15321, 2022.

[88] Chunxue Zuo, Fang Wang, Mai Zheng, Yuchong Hu, and Dan Feng. Ensuring high reliability and performance with low space overhead for deduplicated and delta-compressed storage systems. *Concurrency and Computation: Practice and Experience*, 34:e6706, 2022.

[89] Rakesh Gururaj, Melody Moh, Teng-Sheng Moh, Philip Shilane, and Bhimsen Bhanjois. Performance Centric Primary Storage Deduplication Systems Exploiting Caching and Block Similarity. In *2022 16th International Conference on Ubiquitous Information Management and Communication (IMCOM)*, pages 1–8, Seoul, Korea, 2022. IEEE.

[90] K Venkatesh and D Narasimhan. Revealing the novel precise subset identification and deduplication of audio substance over the shared public environment. *The Journal of Supercomputing*, pages 1–17, 2022.

[91] Keren Jin and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, Haifa, 2009. ACM.

[92] Nimrod Megiddo and Dharmendra S Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Fast*, volume 3, pages 115–130, 2003.

[93] Theodore Johnson, Dennis Shasha, et al. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450, San Francisco, CA, United States, 1994. Citeseer.

[94] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.

[95] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[96] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. CacheDedup: In-line Deduplication for Flash Caching. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 301–314, Santa Clara, CA, 2016. USENIX.

[97] Qianbin Xia and Weijun Xiao. High-Performance and Endurable Cache Management for Flash-Based Read Caching. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3518–3531, 2016.

[98] Jian Liu, Yunpeng Chai, Xiao Qin, and Yuan Xiao. PLC-cache: Endurable SSD cache for deduplication-based primary storage. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, Santa Clara, CA, USA, 2014. IEEE.

[99] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. Nitro: A Capacity-Optimized SSD Cache for Primary Storage. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 501–512, Philadelphia, PA, 2014. USENIX.

[100] Sujesha Sudevalayam and Purushottam Kulkarni. DRIVE: Using implicit caching hints to achieve disk I/O reduction in virtualized environments. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, Goa, India, 2014. IEEE.

[101] Marianne Durand and Philippe Flajolet. Loglog Counting of Large Cardinalities. In *European Symposium on Algorithms*, pages 605–617, Budapest, Hungary, 2003. Springer.

Author's publications

Journals:

1. Amdewar Godavari, Chapram Sudhakar and T. Ramesh, "Hybrid Deduplication System—A Block-Level Similarity-Based Approach" *IEEE Systems journal*, pp. 3860–3870, 2020. DOI: 10.1109JSYST.2020.3012702 (Indexing: SCIE) (**Published**).
2. Amdewar Godavari, Chapram Sudhakar and T. Ramesh,"Hybrid Deduplication System with content-based cache for Cloud" *Future Generation Computer Systems (Elsevier)*. (Indexing: SCIE) *Revision submitted*)
3. Amdewar Godavari, Chapram Sudhakar and T. Ramesh, "File Semantic Aware Primary Storage Deduplication System" *IETE Journal of Research*, DOI: 10.1080/03772063.2022.2050306 (Indexing: SCIE) (**In press**)
4. Amdewar Godavari, Chapram Sudhakar and T. Ramesh, "File aware Distributed Deduplication System in Cloud environment" *Parallel and Distributed Computing (Elsevier)*. (Indexing: SCIE) (*Under Review*)
5. Amdewar Godavari and Chapram Sudhakar, "A survey on Deduplication Systems" *International Journal of Grid and Utility Computing (Inderscience)*. (Indexing: Scopus) (*Under review*)