Popularity-Aware Storage Systems for Big Data Applications

by

Ting Cao

A dissertation submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Auburn, Alabama
August 6, 2022

Keywords: Erasure Code, Cache Replacement, Malware Detection, Popularity Awareness, Big Data Applications, Recommendations

Copyright 2022 by Ting Cao

Approved by

Xiao Qin, Alumni Professor of Computer Science and Software Engineering
Richard Chapman, Associate Professor of Computer Science and Software Engineering
Tao Shu, Associate Professor of Computer Science and Software Engineering
Santu Karmaker, Assistant Professor of Computer Science and Software Engineering

Abstract

Recommendation algorithms play an increasingly dominant role in big data services like Netflix and YouTube. In streaming applications, it becomes unavoidable that trillion tons of personal and industrial data are flooded into the data center. This dissertation is focused on popularity-aware techniques anchored on recommendation algorithms to bolster the performance of data processing. In this dissertation study, we make the following contributions centered around data reconstruction, cache replacement, malware detection, and distributed denial of service (DDoS) detection.

The first contribution of this dissertation is a popularity calculator coupled with a scheduler, where we advocate for erasure-coded data storage systems to archive warm data. Different from hot or cold data, warm data have to be treated in a distinctive way to optimize system performance and storage-space utilization. We employ two machine-learning algorithms to offer online data reconstruction in erasure coded storage systems. We also combine the factors includes the data item size and big data storage location to adjust the popularity value. The final popularity value indicates the malware detection priority. Our system is reliant on a big data storage mechanism to group files into multiple clusters, in each of which files share similar features. Furthermore, we set the prediction module with item size record and storage location which connects the closest users, thereby projecting files that are likely to be accessed in the not-too-distant future. The prediction module is responsible for computing similarities among users so as to set up priority levels of data blocks to be reconstructed. Our experimental results confirm that our system reduces the average waiting time of data recovery while maintaining a high data access performance for on-line users.

The second contribution lies in a popularity-driven cache replacement policy - PDC - catered for big data storage caching systems, in which future accesses predictions are leveraged to push cache-replacement performance to the next level for big data applications. Our PDC governs data recommendation algorithms to gauge popularity values for data objects from active users' access history. Popularity values signify data replacement priorities amid making replacement decisions.

The last contribution of the dissertation study is a similarity-based DDoS detection module. Inspired by a dynamic analysis of access behavior changes in active users, we propose a DDoS anomaly detection model to discover DDoS attack sources by diagnosing users' similarities. The overarching goal of our solution is to pinpoint DDoS by monitoring the similarity of active users around existing users at a low cost. This goal is achieved by our proposed model embracing the following key steps. First, a sample user set is originated. Then, the active users' requests are tracked to assess similarity measures between each active user and sample users. Finally, if the deviation of similarity exceeds prescribed thresholds, detected users will be flagged as anomalous ones.

Acknowledgments

This dissertation would not have been possible without invaluable guidance, help and experience sharing from the people who constantly support and encourage me during the course of my graduate studies at Auburn University.

First of all, I would like to express my sincere gratitude to Dr. Xiao Qin for providing extensive knowledge in the field of computer systems and inexhaustible enthusiasm for research without which this dissertation would not have been possible. While working on the paper titled "Popularity Aware Data Reconstruction (POST)", he gave me numerous constructive advice and suggestions, including setting up strong motivations behind the project, proposing the conceptual and mathematical underpinnings of the POST system, designing the clustering module coupled with a recommendation module, implementing the RS code deployed in POST, to name just a few. By the same token, Dr. Qin shepherded me through the development of the other three research projects centered around popularity-aware data storage services.

I would also like to acknowledge Dr. Wei-Shinn Ku for their participation and engagement. His solid understanding of cloud computing and cyber security, unlimited patience in answering my questions and meticulous working style impressed me in our discussion and meetings. His insightful comments and suggestions helped and enlightened me with literature reviews, appropriate topic targeting, idea extension and demonstration, and experimental validation.

Additionally, I would like to express gratitude to Dr. Jianzhong Huang for his treasured support, which was really influential in shaping my experiment methods and critiquing my results. His extensive experience and strong understanding of large-scale data storage systems

boost my confidence to present mathematical analysis of storage systems, such as property analysis and proofs, in the dissertation.

I would like to thank my committee members, Dr. Richard Chapman, Dr. Shu Tao, and Dr. Shubhra Kanti Karmaker Santu, who reviewed my proposal and dissertation. They gave me insightful and valuable suggestions, by which my dissertation had been immensely improved. I am equally grateful to Dr. Hans-Werner Van Wyk, who gave me helpful comments and suggestions on my dissertation as my university reader.

I would like to thank my friends, lab mates, and research team – Chaowei Zhang, Xiaopu Peng, and Jianzhou Mao for a cherished time spent together in the lab, and in social settings. My appreciation also goes out to my family and friends for their encouragement and support all through my studies. I would like to thank all the computer science and software engineering professors and students, who create and maintain an excellent atmosphere for study and research.

Finally and most importantly, the endless love and support from my family is the most powerful strength that keeps me fighting for my research. My mother Hong Hu, my father Xiaojun Cao, my wife Kan Le and my kids Ellie, Kacie always stay with me, cheering for achievement and overcoming all difficulties.

This dissertation research is made possible by the support from the U.S. National Science Foundation under Grants IIS-1618669 and OAC-1642133.

To my parents and Kan Le.

Table of Contents

Abstract . . . ii

Acknowledgments . . . iv

List of Figures . . . xi

List of Tables . . . xv

1 Introduction and Motivations . . . 1

1.1 Towards A Popularity-Aware Reconstruction Technique . . . 2

1.1.1 Online Recovery of Faulty Nodes in Storage Systems . . . 2

1.1.2 Motivations for Popularity-aware Online Recovering . . . 2

1.1.3 Novel Features and Contributions . . . 4

1.2 Building A Popularity-Driven Caching System for Big Data Applications . . . 5

1.2.1 Cache Replacement Policies . . . 5

1.2.2 Basic Ideas of Popularity-aware Caching . . . 6

1.2.3 Motivations for Popularity-Aware Caching . . . 6

1.2.4 Contributions of the New Popularity-Aware Caching Technique . . . 8

1.3 DDoS Detection Systems for Cloud Data Storage . . . 9

1.3.1 Motivations for Similarity Based DDoS Detection Techniques . . . 10

1.3.2 Contributions for Similarity Based DDoS Detection Technique . . . 10

1.4 Malware Detection in Cloud Storage Services . . . 11

1.4.1 Motivations for a Popularity-Aware Malware Detection Scheduler . . . 12

1.4.2 Contributions of a Popularity-Aware Malware Detection Scheduler . . . 12

1.5 A Road Map . . . 13

2 Related Work . . . 14

2.1 Popularity-Aware Schemes . . . 15

2.2 Data Reconstruction 16
2.2.1 Erasure-Coded Storage Systems 16
2.2.2 Data Archival Schemes 16
2.2.3 Reconstruction Schemes 18
2.3 Cache Replacement 19
2.3.1 Objectives of Caching Systems 19
2.3.2 Traditional Caching Systems 20
2.3.3 Advanced Cache Replacement Strategies 21
2.3.4 What's New in Our PDC? 22
2.4 DDoS Attacks and Detection Methods 22
2.4.1 DDoS Attacks models 22
2.4.2 DDoS Detection Models in Clouds 25
2.4.3 Application Layer DDoS Detection 26
2.5 Malware Detection in Cloud Storage Systems 28
3 A Popularity-Aware Data Reconstruction System 31
3.1 System Architecture 31
3.1.1 Overview 33
3.1.2 Erasure-Coded Storage Clusters 35
3.1.3 The k-prototype Module 36
3.1.4 The Popularity Calculator Module 38
3.2 Designing Basic Modules 40
3.2.1 k-prototype Clustering 40
3.2.2 User-Based Collaborative Filtering 45
3.3 Designing POST 48
3.3.1 k-prototype Controller 48
3.3.2 Reconstruction Controller 50
3.3.3 Time Complexity Analysis 53

3.3.4 Examples 54
3.4 Performance Evaluation 57
3.4.1 Experimental Settings 58
3.4.2 Data archival 60
3.4.3 Space Overhead Analysis 61
3.4.4 Impacts of Similarity 62
3.4.5 Impacts of Number of Users 64
3.4.6 Impacts of Number of User Requests 67
3.4.7 Impacts of Stripe Capacity 69
3.5 Summary 71
4 The Popularity-Aware Cache Replacement 73
4.1 System Design 73
4.1.1 Overview 73
4.1.2 The Popularity Calculator Module 76
4.1.3 Cache Replacement 76
4.2 A Generalized Popularity-Driven Cache Replacement Algorithm 77
4.2.1 User-Based Collaborative Filtering 77
4.2.2 A Popularity-Based Cache Replacement Policy 82
4.3 Performance Evaluation 85
4.3.1 Performance Metrics and Experimental settings 85
4.3.2 Overhead Analysis 86
4.3.3 Hit Ratio Analysis 88
4.3.4 Byte Hit Ratio Analysis 90
4.3.5 Impacts of Recommendation List Length 91
4.3.6 Comparison with the Advanced Cache Replacement Strategies 94
4.4 Summary 95
5 Similarity-Based DDoS Detection 97

5.1 An Overview 97
5.2 A Similarity-based DDoS Detection System 99
5.2.1 The Basic Idea 99
5.2.2 High-Level System Architecture 100
5.2.3 Concepts and Key Steps 101
5.3 Algorithm Design 105
5.3.1 Overhead Analysis 107
5.4 Performance Evaluation 108
5.4.1 Evaluation of Overhead 108
5.4.2 Evaluation of Accuracy 109
5.5 Summary 110
6 Popularity-Aware Malware Detection 112
6.1 Basic Idea 113
6.2 System Architecture 114
6.3 Concepts and Main Steps 115
6.4 Algorithms 117
6.5 Summary 119
7 Conclusions and A Future Research Plan 121
7.1 Conclusions 121
7.1.1 Data processing and scheduling 121
7.2 Advanced algorithm exploration 122
7.3 A Future Research Plan 123
7.3.1 Research Direction 1 123
7.3.2 Research Direction 2 123
7.4 Summary 124
Bibliography 126

List of Figures

2.1 The high-level DDoS attack procedures 23

2.2 Main stream DDoS attacks in cloud computing platforms. 25

2.3 The high-level procedures of static and dynamic malware detection schemes. . . 29

3.1 The architecture of POST, when warm data with features comes from 3X replication storage system, the Clustering part will archive data to erasure code based storage system by k-prototype algorithm. When users are accessing data with data block failures, recommendation part offers a recommendation list based on users set, and give a reconstruction list for each stripe in the storage system. . . 32

3.2 Generating parity blocks in (k+r,k) Reed-Solomon codes. 35

3.3 The layout in a conventional RS-coded storage cluster. 36

3.4 The work flow of k-prototype module. 37

3.5 The work flow of recommendation module. 40

3.6 Data objects are arranged in a way that, in most cases, data objects placed on one stripe belongs to the same cluster. Objects $A1 - A4$ form the first stripe stored across $disk1 - disk4$. A large cluster may be handled by multiple stripes. Objects $B1 - B7$ are stored in two strips - $(B1 - B4)$ and $(B5 - B7)$, which are archived on $disk1 - disk4$ and $disk1 - disk3$, respectively. Objects $C1 - C8$ are placed among three stripes. 55

3.7 The case for k-prototype. The total waiting time as 13 with an average of 3.25 time unit per user. 55

3.8 The case for recommendation list. The total waiting time as 11 with an average of 2.75 time unit per user. 56

3.9 The case for POST. The total waiting time as 10 with an average of 2.5 time unit per user. The waiting times of the four users are 2, 3, 1, and 4, respectively. . . 57

3.10 Distribution of user ratings with respect to movie IDs. Approximately 60% of ratings are placed on items, the movie IDs of which are smaller than 3,500. . . 58

3.11 distribution of user ratings for the movies dataset 59

3.12 The impacts of the number of clusters on data archiving time in the POST system. The number of iterations, the number of data nodes, the number of parity nodes, data block size are configured to 30, 4, 4, 64 MB, respectively. The archived dataset contains movie data objects of approximately 25 TB. 61

3.13 Impact of weight value on prediction accuracy. The weight value is proportional to the prediction accuracy. The correlation between weight value and prediction accuracy guides system administrators to optimize the h value in the recommendation list $P(u_x, h)$ (see Eq. 3.14. 62

3.14 The impact of prediction accuracy on average waiting time. There are 100 users concurrently accessing 1,000 data objects. Boosting the prediction accuracy shortens waiting time. 63

3.15 The average waiting time measures of the six reconstruction strategies managing the movie datasets and book datasets. The number of users varies from 20 to 1000 in movie dataset and 20 to 800 in book dataset. 65

3.16 Average waiting times of the six reconstruction strategies. The number of requested data objects per user varies from 5 to 100 with an increment of 5. The number of users is set to 100. 67

3.17 The average waiting times of the four reconstruction strategies. The number of data objects stored in one stripe varies from 1 to 2,048. 70

4.1 The high-level architecture of the PDC system. 74

4.2 The work flow of PDC cache replacement. 79

4.3 The impacts of cache size on the hit ratio performance of the three cache replacement policies. The cache size ranges from 2 GB to 2048 GB; the number of data objects is set to 200, 500, 1000, 2000, 4000, and 8000. 89

4.4 The impacts of cache size on the byte hit ratio performance of the three cache replacement policies. The cache size ranges from 2 GB to 2048 GB; the number of data objects is set to 200, 500, 1000, 2000, 4000, and 8000. 90

4.5 The impacts of recommendation-list length on the average time delay and hit ratio performance of the PDC cache replacement policy. The cache size is set to 32GB; the recommendation list is set to 10, 20, 40, 60, 80, and 100; the number of data objects is set to 200, 1000 and 4000. 92

4.6 The impacts of cache size on the hit ratio performance of the PDC and two advanced technology cache replacement policies. The cache size ranges from 2 GB to 2048 GB; the number of data objects is set to 1000, 2000, 4000, and 8000 94

5.1 The architecture of similarity-based DDoS detection model running in clouds. abnormal users are identified through the similarity comparison. 100

5.2 The main steps to check if the current user is abnormal. 101

5.3 The time cost vs. an increasing size of the training set (Users History Behavior). 109

5.4 Detection Accuracy with the an increasing number of data items. 110

6.1 The architecture of a malware detection scheduler running in clouds. Popular data objects receive a high priority to be scanned earlier than unpopular data objects. 114

List of Tables

I Qualitative comparisons among data storage systems that are closely related to POST. 17

II The existing DDoS detection techniques 26

I Symbols and Notation 41

II sample recommendation list 56

III sample recommendation list 57

I Symbols and Notation 78

I A list of candidate recommendation algorithms are readily plugged into the popularity-aware malware detection scheduler. 116

Chapter 1

Introduction and Motivations

The phenomenal growth in streaming and web services (e.g., Netflix and YouTube) gives rise to a sharp spike in demands to construct big data centers housing high-end clusters embracing tens of thousands of storage nodes. In streaming applications, it becomes unavoidable that trillion tons of personal and industrial data are flooded into the data center. For example, 400 billion photos were stored on Facebook servers in Feb 2014 [85]; IDC reports that from 2005 to 2020, the global data volume grew by a factor of 300, from 130 exabytes to 40,000 exabytes, representing a double growth every two years [40]. Moreover, cloud computing enables users to access information from anywhere and at any time on any device [45]. Therefore, big data accompanied by its high growth rate makes it critical and indispensable to optimize the performance and scalability of data storage systems.

In this dissertation we exploit the following four research areas to spearhead new popularity-aware techniques to bolster reliability, performance, DDoS attack protection, and malware detection.

- A popularity-aware reconstruction technique in erasure-coded big-data systems (see Section 1.1).
- A popularity-driven caching system for big data applications (see Section 1.2).
- DDoS detection systems for cloud data storage (see Section 1.3).
- Data security and malware detection in cloud storage services (see Section 1.4).

The road map in Section 1.5, at the end of this chapter, gives a good overview of this dissertation's organization.

1.1 Towards A Popularity-Aware Reconstruction Technique

1.1.1 Online Recovery of Faulty Nodes in Storage Systems

It is demanding to build modern storage systems to meet requirements of big-data applications. In data centers, about 90 percent of data have been created in the last two years [1]; for example, 400 billion photos are stored on Facebook servers in Feb 2014 [85]. Fault tolerance is an indispensable key technique in large scale distributed storage systems. For instance, there are on an average of 50 machines, 95500 blocks, and 180 TB of data to be recovered each day in Facebook [85]. Recent studies advocate for replication and erasure coding to maintain data durability [120]. Data replication has become a de facto standard; for example, the Hadoop distributed file system [109] and the Google file system [41] utilize the three-way replication storage policy. Replication, of course, provides good reconstruction efficiency at the cost of high storage space consumption. In contrast, erasure code can save storage space while paying extra cost to calculate parity. To alleviate the high computing-time cost of erasure coded storage, we propose, in the first part of the dissertation research, a reconstruction technique called POST to reduce user waiting times when a storage system is carrying out data reconstruction while responding to user requests.

Based on the access frequency, the data can be differentiated as hot, warm and cold. In general, compared to the content in hot data, warm data has a lower request rate. Warm data in turn is accessed more frequently than data in cold storage systems.

1.1.2 Motivations for Popularity-aware Online Recovering

POST is responsible for the online recovery of faulty nodes in a storage system when the system is actively servicing user I/O requests. The following four facts motivate us to contrive POST.

- Warm data has been generated at a fast pace in recent years.

- Erasure-coded storage systems are an efficient and fault-tolerant technique to store large-scale data.
- It is a common practice to predict future data popularity from historical accesses and user preferences.
- There is the lack of data reconstruction techniques aiming to reduce waiting times from the perspective of online users.

Motivation 1. The volume of warm data has been growing rapidly in recent years; importantly, warm data account for more than 80 percent of all data types in combine [85]. The access frequencies of warm data are anywhere between those of hot data and cold data, implying that warm data are somehow occasionally accessed. To maintain data fault tolerance in real-world applications, data centers manage hot and cold data using the *3x replication* [23] and *erasure code* [70] techniques respectively. When it comes to storing and accessing warm data, one ought to address the concerns of storage space and reconstruction speed in a holistic way.

Motivation 2. Erasure code techniques have been widely adopted in archival storage [113], data centers [115], and cloud storage systems [19]. For example, the Reed-Solomon (RS) [100] code is a popular erasure code scheme for data encoding and decoding. The RS code achieves a fault-tolerance level higher than that of XOR- based parity array coding [49]. In this study, we put a spotlight on erasure-coded storage systems tailored for a massive amount of warm data.

Motivation 3. Data mining - an established cutting-edge technique - has been applied to make predictions on customers' preferences [97]. Most popular websites (e.g., YouTube and Netflix) dynamically manage a preference list for each user to predict content to be accessed by the user. This technological trend motivates us to devise a data-mining-based recommendation module to govern a data reconstruction process in storage systems. Corrupted data

that are likely to be accessed in the not-too-distant future should be recovered in an early stage to alleviate adverse impacts of faulty storage nodes.

Motivation 4. Ideally, a data-reconstruction system should achieve high reliability, low storage cost, and fast reconstruction speed. From the perspective of users, superb reliability and short response time are major concerns when data stored in a system are being accessed. Intuitively, boosting reconstruction speed leads to short response time or waiting time in storage systems. Apart from improving reconstruction speed, we exploit a data-mining-based approach to slash waiting times when an erasure-code storage system is carrying out data reduction. To the best of our knowledge, popularity-aware data reconstruction in storage systems is still in its infancy. To fill this technology gap, we develop a recommendation module to predict the likelihood of future data accesses from content popularity.

1.1.3 Novel Features and Contributions

POST seamlessly integrates two distinctive modules, namely, data archiving and online reconstruction. Recognizing that a majority of data's temperatures tend to change from hot to warm, the data archiving module is in charge of converting warm data from three replicas into the erasure-coding format. A salient feature of this module relies on a classification scheme that groups relevant data into a single erasure-coded stripe. The second module governs online data reconstruction for erasure-coded storage created by the data-archival module. Furthermore, POST predicts user preferences, which drive the data-reconstruction priority. Put another way, popular corrupted data are recovered by POST prior to repairing non-popular data. POST maintains a data-reconstruction queue, in which data with high access probability receive a high priority during the course of online data reconstruction.

In this study, we devise the POST system to classify data when the data are archived, followed by generating a prediction list. In the clustering module of POST, K-means and K-modes along with the text analysis are incorporated in the POST system. We develop a user-based collaborative filtering recommendation algorithm to guide the data reconstruction

procedure. With recommendations in place, popular data will be recovered first, prior to non-popular counterparts. The results from the trace-driven experiments indicate that POST is adroit at cutting back user waiting time when an erasure-coded storage system is carrying out online data reconstruction. The main contributions of this work are summarized as follows:

- We articulate the design of the POST system, which offers a storage-space-efficient solution for large-scale data centers.
- We incorporate the K-prototype and naive-Bayes-based text-analysis algorithms to cluster files during the data archival stage.
- We develop a recommendation system to favor popular data over non-popular ones during the course of data reconstruction.
- We replay real-world traces to evaluate the performance of the POST system.

1.2 Building A Popularity-Driven Caching System for Big Data Applications

1.2.1 Cache Replacement Policies

Caching, among cutting-edge I/O techniques, is a prominent way of improving the overall performance of storage systems [61]. Caches have proven to be vital mechanisms for reducing long latency of disk accesses; the effectiveness of caching is significantly influenced by data replacement policies. A cache serves the most popular data items without querying the back-end storage nodes, ensuring uniform I/O load among the back-end nodes [34]. Meanwhile, the I/O performance gap between data storage and main memory systems has been widened in the past decade [82]. In large-scale data storage clusters, caching reduces user-perceived delays as well as data loading time. In the realm of big data processing, it remains a grand challenge to make optimal cache replacement decisions on which cached data should be evicted. Nevertheless, existing replacement policies have been devised in

a heuristic way. Predominant policies, including the least recently used policy (LRU) [39] and the least frequently used policy (LFU) [107], are top contenders across the board under various workload conditions.

1.2.2 Basic Ideas of Popularity-aware Caching

We propose in this study a popularity-aware caching policy referred to as *PDC*, which leverages future accesses predictions to push cache-replacement performance to the next level for big data applications. Our PDC advocates for data recommendation algorithms to gauge popularity values for data objects from active users' access history. Popularity values in turn signify data replacement priorities amid making replacement decisions. In addition to orchestrating active users (e.g., log into the system) and inactive users (e.g., log out from the system), a management module in the PDC system keeps track of access preferences (e.g., popularity measures) of the users.

1.2.3 Motivations for Popularity-Aware Caching

The following three facts motivate us to contrive the PDC system.

- In recent years, large-scale storage systems have been used increasingly to store complex digital data.
- Storage systems embrace caching techniques to speed up I/O performance in terms of loading data objects.
- It is viable and practical to predict future data popularity from historical accesses coupled with user preferences.

Motivation 1. Big data has become a frontier topic for computer systems developers and researchers because the growing exploitation of audio/video streams is generating a massive amount of complex and pervasive digital data. Also, big data analysis largely depends on the effective mining of massive datasets at multifaceted levels including modeling, visualization,

prediction, and optimization. It is nontrivial to acquire and integrate an enormous amount of data from distributed locations. For instance, more than 175 million tweets containing text, image, video, or social relationship are generated by millions of accounts distributed globally [46]. Importantly, with the advent and accessibility of modern cloud computing infrastructures, cutting-edge technology available to the public has turned the structure of data towards interconnected and rapidly growing ones. When it comes to storing and accessing big data objects, there is a pressing demand to optimize I/O performance of storage systems in a holistic way.

Motivation 2. Storage clusters - built from the off-the-shelf commodity storage nodes - address the growing needs of big data applications. To improve the performance of large-scale storage clusters for active users to access, the caching technique has been paid tremendous attention in the past decade. Large cache space residing in a storage cluster allows data to be read and written at a high speed. Due to the limited size of the cache, there is a preference in keeping frequently accessed data objects. When a caching buffer is full, a cache replacement policy has to be kicked in to evict data to release space for new incoming data objects. The cache replacement mechanism is responsible for removing unwanted data objects that are likely to induce cache pollution and poor I/O performance. Since the demands and characteristics of big-data applications always vary with time, the existing caching schemes become inadequate for large-scale storage systems [38]. The commonly adopted cache replacement policies like the least recently used (LRU) [39] and least frequently used (LFU) algorithms [107] fail in accurately predicting data objects that have the highest possibility to access the not-too-distant future, because the existing policies passively make use of historical access patterns to speculate popular data. To fill this technology gap, we propose a popularity-driven caching system or PDC that makes the most appropriate data-replacement decisions through popularity assessments by the virtue of data mining.

Motivation 3. Recent evidence suggests that it is viable and practical to predict future data popularity from historical I/O accesses coupled with user preferences. For instance,

data mining - an established cutting-edge technique - has been applied to make predictions on customers' preferences [97]. This technological trend motivates us to devise a data-mining-based recommendation module in our PDC system to govern the cache replacement process in storage systems. In PDC, we advocate for the data mining technique to speculate future popular data objects that deserve being cached while evicting unpopular ones. The prominent benefits of this design philosophy are three-fold. First, PDC proactively predicts future I/O access patterns rather than relying on a conventional passive approach. Second, PDC exploits statistic analysis to select the most appropriate data objects to be kept in the cache. Third, PDC leverages comprehensive assessments to optimize the cache replacement performance.

1.2.4 Contributions of the New Popularity-Aware Caching Technique

To offer an advanced cache replacement solution for large-scale data centers, we design and evaluate the PDC system performing the role of a caching module for storage clusters. It is noteworthy that storage clusters has become a predominant computing platform in data centers, because multiple storage nodes offer high data access rates by the virtue of parallel I/Os. From the perspective of system architecture, the cache region of PDC is furnished by distributed main memory in a large storage cluster. While users are accessing big data objects, PDC is responsible for governing the cache replacement. Upon the arrival of a user request, requested data might be missing from the cache. In this case, PDC meticulously determines if the requested data items ought to be kept in the memory caches while evicting another object to release cache space. The data to be evicted from the cache must be judiciously picked by PDC to maintain a high cache hit rate.

We devise a recommendation module in PDC to favor popular data over non-popular ones during the course of cache replacement. This module is adroit at predicting future user accesses derived from active users' access preferences accompanied by personal interests. We quantify the level of interest using user ratings history logs, which are harvested and managed

in the recommendation module. The recommendation module generates a popularity list of data objects, thereby stipulating the replacement priorities of all the cached data objects.

To quantitatively and systematically evaluate the performance of PDC, we replay real-world traces to mimic the behavior of real-world storage clusters housed in data centers. We simulate scenarios where PDC orchestrates the cache space storing movie data objects acquired from a real-world data set. The keep metrics used to gauge the performance include hit ratios and byte hit ratios. Furthermore, we configure cache size and cache replacement policy as the system parameters in PDC. To demonstrate the strengths of PDC, we compare our design with the conventional cache replacement policies like LRU and LFU.

In a nutshell, the main contributions of the second part of the dissertation study are summarized as follows:

- We articulate the design of the PDC system, which offers a cache replacement solution for large-scale data centers.
- We develop a recommendation module to favor popular data over non-popular ones, thereby enhancing the caching performance.
- We replay real-world traces accessing movie data to compare the performance of the PDC system with the existing solutions.

1.3 DDoS Detection Systems for Cloud Data Storage

We devise the DDoS detection model that seamlessly integrates two distinct modules - a sample user generator and a similarity comparison module. The first module elects a group of high fitness users from all the legal users to build a sample user set, which is periodically updated by our system. The second module is in charge of calculating the similarities among active users and sample users. We treat a user as an abnormal user if the user's similarity measures are dramatically changed with a high percentage.

In what follows, Section 1.3.1 emphasizes the motivations for designing DDoS detection systems equipped with similarity measurement techniques. Section 1.3.2 summarizes the contributions made in our design and development of simiarility-based DDoS detection techniques.

1.3.1 Motivations for Similarity Based DDoS Detection Techniques

Two reasons motivate us to advocate for the similarity-based DDoS detection strategy. First, online cloud servers are accessed and shared by a large-scale user pool. Users' historical access records offer excellent samples to discover outlier users such as bots. Second, when bots attempt to simulate human users, it is a challenge to find attack sources with a low overhead. If a DDoS detector fails in swiftly pinpointing attackers, any detection delay may pose potential security threats to the entire system and users. Our DDoS detection model ensures that active users' access behaviors pass through a thorough yet lightweight analysis.

Below is the list of two motivations for this part of the dissertation research.

- Motivation 1: Historical access records registered in online cloud servers offer sampling points to sense outlier users.
- Motivation 2: DDoS-detection delays inevitably pose security threats to online systems and users.

1.3.2 Contributions for Similarity Based DDoS Detection Technique

We present the design of DDoS detection model to optimize a malware detection sequence of data objects to prioritize high-risk data in cloud storage systems. The overarching goal of the similarity comparison module is to glean users' current and historical request records, followed by calculating similarity measures of active users and the sample ones. More specifically, the similarity module illustrates the users' behavioral changes with respect to user requests. This module works in full capacity to dynamically track and monitor the active users' similarity metrics to sense any abnormal behaviors.

A summary of the contributions made in the third part of the dissertation research is listed below.

- We spearhead the design of the similarity-based DDoS detection system, which offers a low overhead solution for online big data storage DDoS detection.

- We develop a sampling module for user selections in hope of collecting a high fitness user set supporting similarity comparison, thereby further curtailing detection I/O cost and boosting detection accuracy.

- We replay two representative real-world traces accessing the data streams to evaluate the overhead with different sizes of the training sets and the detection accuracy of the similarity-based DDoS detection model with the existing solutions.

1.4 Malware Detection in Cloud Storage Services

In this section, we discuss the motivations for designing popularity-based malware detection systems in Section 1.4. Section 1.4.2 boasts the contributions of our dissertation research centered around the design of popularity-aware malware detection techniques.

To speed up malware detection in cloud storage, we devise a popularity-aware malware detection system that seamlessly integrates two distinct modules - a popularity predictor and a malware detector. The first module makes data popularity prediction possible by adopting the *user-based collaborative filtering* algorithm or UBCF [133][47]. The second module is in charge of detecting malware in data objects on the basis of future popularity. Hot data that are likely to be frequently accessed receive a high priority to go through malware screening. With the assistance of the popularity predictor, the malware detector ensures that data are malware free before being retrieved.

1.4.1 Motivations for a Popularity-Aware Malware Detection Scheduler

There are two main reasons that motivate us, in the fourth part of the dissertation research, to devise a popularity-driven malware detection framework, in which popular data are a given high priority amid malware detection procedures.

- Popular data are accessed by a large number of users.
- Popular data are likely to be accessed in the not-too-distant future, making it urgent to identify malware from the popular data.

Motivation 1. When a popular data object is approved to be malware-free, all requests accessing the data are protected from malware. Any malware-free data that are popular can immediately benefit a large group of users.

Motivation 2. If malware detection is not carried out on popular data in a timely manner, the data may pose potential security threats to users. Our proposed malware detection scheduler ensures that popular data pass through a thorough malware detection earlier than unpopular ones.

1.4.2 Contributions of a Popularity-Aware Malware Detection Scheduler

We develop a recommendation system to favor popular data over non-popular ones during the course of malware detection. The overarching goal of our scheduler is to optimize a malware detection sequence of data objects by deploying a machine-learning-enabled recommendation algorithm to prioritize high-risk data in cloud storage systems. More specifically, a UBCF-based management module periodically sorts data objects according to popularity. The schedule works in full capacity to dynamically set up the priorities of data objects with respect to popularity measures.

Let us list the key contributions of the last component of this dissertation study here.

- We deploy a novel recommendation system to favor popular data over non-popular ones.

- We devise a popularity-based scheduler to handle malware detection requests, aiming to maximize system security.

1.5 A Road Map

The remainder of this dissertation is organized as follows. Chapter 2 surveys the related work of popularity-aware modules and existing solutions of our proposed models without popularity-aware techniques. The organization, algorithms, and performance evaluation of the Popularity-aware data reconstruction model are described in Chapter 3. The organization, algorithms, and performance evaluation of the Popularity-driven cache replacement strategy and similarity-based DDoS detection technique are proposed in Chapter 4 and Chapter 5, respectively. A popularity-aware scheduler for malware detection in big data systems including the system architecture, the algorithms are illustrated in Chapter 6. We conclude this dissertation in Chapter 7.

Chapter 2

Related Work

This chapter is dedicated to the prior studies that are closed related to this dissertation, which is centered around data reconstruction, cache replacement, DDoS detection, and malware detection schemes. We kick off this chapter with the literature review focused on popularity-aware schemes, as four popularity-driven systems designed in Chapter 3, Chapter 4, Chapter 5 and Chapter 6 share a common and vital module: a popularity calculator. Next, we survey leading-edge techniques from the perspectives of data storage and data reconstructions, including erasure-coded storage systems, data archival schemes, and data reconstruction techniques. Then, we continue the literature review by shifting our attention to cache replacement policies, where we elaborate on objectives of caching systems, traditional caching strategies, advanced caching techniques, as well as the new features of our design. This chapter also introduces the background knowledge about DDoS attacks and detection methods: we cover the surface of DDoS attack models, detection solutions in clouds, and application-layer DDoS detection techniques. Finally, the related work of malware detection in cloud storage systems is presented as the last piece of the chapter.

More specifically, we organize this chapter in the following way. Section 2.1 reveals the existing popularity-aware schemes for data processing. Section 2.2 illustrates a selection of previous studies focused on erasure-coded storage systems, data archival and data reconstruction schemes. Section 2.3 describes the workload and differences between traditional and advanced cache replacement strategies. The related work of DDoS attacks and detection can be found in Section 2.4. In Section 2.5, we introduce the representative solutions of malware detection methods customized for cloud storage systems.

2.1 Popularity-Aware Schemes

Previous studies have demonstrated that popularity-aware techniques are quite promising in boosting caching performance in storage systems. For example, Jin *et al.* devised a GDS-Popularity (GDSP) cache management algorithm, which effectively captures and maintains an accurate popularity profile of web objects requested through a caching proxy [56]. Emre and Deniz proposed distributed caching policies that incorporating the mobility of users as well as content popularity when delay deadlines are either below a certain threshold or relaxed [88]. Ali *et al.* utilized the support vector machine (SVM) and decision tree techniques to discover web-proxy log files, followed by predicting whether or not the classes of objects will be re-visited. The findings confirmed that such predictions have the full potential to brush up the performance of the cache system.

Apart from optimizing caching performance, dramatically boosting the I/O performance of data storage systems is a well-received benefit of popularity-aware schemes. For instance, to efficiently avoid wasting cache space for storing on-path content duplicates, *PPCS* is intended to cache chunks according to content popularity in a way that improves cache diversity [87]. Xie *et al.* developed a popularity-aware scheduling algorithm or *PAS* [124] for network coding, where a popularity value is automatically created when a network node encodes incoming blocks to generate a new one. PAS schedules priority-enabled blocks in accordance with associated popularity values, thereby speeding up content dissemination and improving information transmission efficiency.

The above popularity-aware techniques are mainly focused on optimizing I/O and parallel computing processes by advocating for data popularity concept. Unlike the existing popularity-aware schemes, our PDC has a salient strength in leveraging a user-based collaborative filter to facilitate the popularity-aware cache replacement policies.

2.2 Data Reconstruction

Recent evidence shows that I/O latency and storage cost can be jointly minimized by considering three dimensions: erasure code selections, encoded-chunk placement, and scheduling policies [123]. These three dimensions have been explored in the context of erasure coded storage systems, data archival techniques, popularity-aware schemes, and data reconstruction strategies. Let us shed some light on the related studies categorized into the four core areas, namely, erasure-coded storage (see Section 2.2.1), data archival (see Section 2.2.2), popularity awareness (see Section 2.1), and data recovery (see Section 2.2.3).

2.2.1 Erasure-Coded Storage Systems

Table I summarizes the major differences between our proposed POST and the existing big data storage systems. The robustness of data storage systems can be improved by either applying file duplication or erasure code techniques. For instance, HDFS [15] augments 3x replicated files to enable data recovery. PARIX [68] transforms the original formula of parity calculation, making use of data deltas (i.e., between current and original data values) instead of parity deltas to obtain parities during journal replay. A novel erasure code technique called local reconstruction codes or *LRC* was implemented in Windows Azure Storage (a.k.a., WAS) [48]. Facebook's Warm BLOB storage system [85] embraces Reed-Solomon coding and lays blocks out on different racks to ensure resilience to disks, machines, and rack failures within a data center.

2.2.2 Data Archival Schemes

When it comes to archiving a sheer amount of data, one has to construct archival storage systems accompanied by data archival techniques. Archival source is originated from newly created data; such archival storage systems are featured with strong fault tolerance. Modern archival storage systems are designed to store big data using erasure codes to improve space

Table I: Qualitative comparisons among data storage systems that are closely related to POST.

storage system	**Erasure Coded Storage**	**Data Recovery**	**Data Archival**	**Popularity Awareness**
HDFS [15]	$\times$	$\surd$	$\surd$	$\times$
PARIX [133]	$\surd$	$\surd$	$\times$	$\times$
WAS [48]	$\surd$	$\surd$	$\times$	$\times$
BLOB [85]	$\surd$	$\surd$	$\surd$	$\times$
CF-hadoop [133]	$\times$	$\surd$	$\surd$	$\surd$
POST (This Study)	$\surd$	$\surd$	$\surd$	$\surd$

efficiency. Archival sources have replicas stored in disks, where data archival techniques migrate the data replicas into erasure-coded archival formats.

A raft of erasure-coded archival systems have been proposed in the past decade. For example, the erasure code scheme was adopted by the following four archival storage systems. Pergamum is an energy-efficient disk-based archival storage system [113]. Cleversafe is a cost-effective storage system for actively archiving data [26]. Tahoe-LAFS is a decentralized storage system, which offers provider-independent security for long-term data storage [121]. HDFS-RAID is a HDFS module deploying RS codes to store old data sets that are accessed by a limited number of jobs in Hadoop [115]. aHDFS is an erasure-coded data archival system dedicated to archiving unpopular data stored on Hadoop clusters [23].

Different from the aforementioned archival schemes, our POST, customized for multimedia storage systems, leverages a recommendation algorithm to optimize reconstruction performance. POST places data sharing similar features on the same data stripe, thereby creating ample opportunities to optimize data reconstruction scheduling decisions to slash user waiting time.

2.2.3 Reconstruction Schemes

There is a pressing need to optimize data reconstruction in RS-code-based storage systems. Representative optimization schemes include, but not limit to, maximally recoverable local reconstruction code (i.e., *MRLRC*, *DLRC*, *AZ-code*) [43] [81] [125] , proactive reconstruction I/O for erasure-coded storage clusters (i.e., *PUSH*) [49], decision-tree-based reliability modeling [71], and Mojette-transform-based erasure correction codes(i.e., *LDPC*) [11]. We classify these approaches into the following four categories.

- *Reducing the number of required decoding nodes.* MRLRC separates data blocks into small groups in a way that local parity blocks are fabricated to reduce the number of needed blocks. Such small local groups make it possible to reconstruct data within local groups rather than global groups [43].

- *Exploiting lightly-loaded nodes.* PUSH utilizes proactive reconstruction I/Os, in which decoding loads are evenly dispersed among lightly-loaded nodes to dramatically improve reconstruction performance by eliminating the bottleneck in replacement nodes [49].

- *Recovering data blocks in advance.* In the decision-tree-based modeling mechanism, decision trees and gradient boosted regression trees are incorporated to predict failures by exploiting features like temperatures, uncorrectable errors, and spin up time [71].

- *Reducing complexity of matrix transform.* Matrix transform is a bottleneck in computing erasure code. LDPC substitutes mojette transform for common vandemonde matrix transform to mitigate the expensive matrix transform cost [91].

Compared with the above reconstruction schemes, POST scheme has three salient features. First, POST is data construction technique tailored for node-based storage clusters by exploiting erasure code. Second, POST advocates the machine learning algorithms for

clustering data and scheduling reconstruction requests. Third, POST induces fairly low computation overhead thanks to the fact that clustering archival and recommendation lists are processed prior to data reconstruction.

2.3 Cache Replacement

2.3.1 Objectives of Caching Systems

Growing evidence shows that the performance of caching systems can be jointly boosted from three dimensions, namely, cache consistency, cache pre-fetching, and cache replacement [35] [37] [63]. The primary objective of an ideal cache replacement policy is to evict undesired objects to enhance cache utilization. Cache replacement strategies aim to optimize cache hit rates and mitigate heavy I/O loads on storage systems.

Important factors of data objects affecting cache replacement efficiency are summarized as follows.

- Recency: time of the last reference to an object
- Frequency: number of requests to an object
- Size: data object size
- Cost of fetching the object: cost to fetch an object from its origin server
- Modification time: time of last modification

Cache replacement policies can fall into four categories, namely, (1) key-based replacement strategies, (2) function-based replacement strategies, (3) randomized-based algorithm, and (4) weighting-based algorithm. [95]

Most of the existing cache replacement strategies are focused on detecting frequency, recency, and modification time to speed up performance (see, for example, SIZE, GDS, LRU, LFU). Hit ratio and byte hit ratio are key metrics to gauge the performance of cache

replacement strategies. Our PDC has an edge over the existing solutions in terms of these performance metrics because PDC gives a boost in the accuracy of frequency detection by the virtue of data popularity values.

2.3.2 Traditional Caching Systems

In general, a fetch policy governs the appropriate time at which information is brought into the cache. Fetch policies are categorized into two camps, namely, demand fetching and pre-fetching schemes [78]. In this study, we make a decisive move to seamlessly integrate our cache replacement algorithm with demand fetching rather than pre-fetching ones. The reason for this design decision is three-fold. First, demand prefetching enables our system to utilize large amounts of resources in support of big data applications. Second, our system enjoys less loading latency during the startup phase. Third, fetching policies are independent of cache replacement techniques, implying that our replacement algorithm can be readily blended with pre-fetching counterparts. Classic cache replacement policies exhibit unique advantages. For example, SIZE treats object size as a primary factor during the course of caching objects. When a cache is full, SIZE proactively evicts large objects to release excessive free cache space. SIZE is facing the "cache pollution" issue because small objects are residing in the cache with less likelihood of being accessed. SIZE is inapplicable for storage systems where data objects are identical in size. The greedy dual-size algorithm or GDS keeps track of object size, access latency, as well as cost. GDS is a value-based cache replacement policy; while making replacement decisions, GDS replaces objects that have the smallest cache value. Unlike SIZE and GDS, PDC takes advantage of popularity as cache values to elect the most appropriate objects removed from the cache. In contrast to the traditional cache replacement strategies, PDC is a popularity-aware technique catered to big-data online services.

A raft of cache replacement algorithms have been proposed in the past decade, attempting to maximize cache hit ratio. For example, greedy-dual exploited naive Bayes, decision

tree, and support vector machine techniques to improve the byte hit ratio and hit ratio metrics [8]. Jain *et al.* optimized cache performance through the Belady's algorithm [52]. Jaleel *et al.* proposed a re-reference interval prediction based cache replacement policy [53]. Sheu *et al.* devised wildcard rules caching algorithm and a rule cache replacement (RCR) algorithm by the virtue of the cover-set approach [108].

A handful of intriguing studies have conducted to significantly cut back I/O latency and energy consumption. For instance, Moriyama *et al.* proposed an efficient congestion control scheme to retrieve distributed chunks from multiple nodes in information centric networking [132]. Li *et al.* developed a load-balanced cache placement algorithm by expanding the Bayesian networks [66]. Li *et al.* presented a delay-constrained sleeping scheme, which aims to conserve energy in a cache-aided ultra-dense network [72].

The traditional caching systems are passive, because these systems rely on access history as well as users' preferences to predict future accesses. In thr realm of big data storage systems, the preferences of online users are constantly changing, the predictions made by conventional caching replacement policies become inadequate.

2.3.3 Advanced Cache Replacement Strategies

Existing advanced cache replacement techniques can be categorized into two groups. In the first category, intelligent algorithms are independently deployed; the second one advocates for combining the intelligent algorithms with cache replacement policies. Both approaches are reliant on the predictions of future access probabilities to enhance cache performance. Recently, the machine learning algorithms, such as artificial neural networks (ANNs), support vector machines (SVMs), the decision tree algorithm, and naive-bayes-based prediction algorithms, have been widely employed in conventional cache replacement policies to optimize the performance. For example, Huang et al. proposed a Markov chain-based predictive model, which was incorporated in the predictive greedy dual-size frequency artificial intelligence (PGDSF-AI) policy [50]. Semantic analysis concepts like semantic segments,

probe queries, and remainder queries were used to construct a segment access-aware semantic cache, which predicts the items that are likely to be accessed by users in the future to improve cache hit ratio in [75]. Most of the existing machine learning algorithm-based caching systems embrace complicated, accurate feature selection and training processes. Generally, these systems are executed in offline components that require extra I/O resources.

Due to dynamic changes of active users, on the other hand, the popularity of data objects is shifted accordingly. With low-cost dynamic popularity monitoring in place, our PDC keeps track of data-object popularity information through active users' preferences. Each login or logout action may spark a change in cache replacement priorities.

2.3.4 What's New in Our PDC?

In contrast to the aforementioned cache management strategies, our PDC proposed in Chapter 4 orchestrates a recommendation algorithm to deliver superb cache replacement performance for storage systems in general and multimedia storage systems in particular. Unlike the existing solutions, PDC replaces data in the cache according to popularity values. therefore, ample opportunities are fostered in PDC to make optimal cache replacement decisions that lead to improved hit ratio.

2.4 DDoS Attacks and Detection Methods

2.4.1 DDoS Attacks models

Distributed Denial-of-Service (DDoS) attack - an advanced form of DoS attack - is a coordinated attack against one or multiple targets through a group of computers [112]. DoS attacks are categorized into the network layer attacks, application layer attacks, data flooding attacks, and protocol-feature-based attacks [33]. All the types of DoS attacks aim to exhaust system resources at various layers in computing systems. For example, a network layer attack attempts to crash routers through buffer overrun errors in the password checking routine.

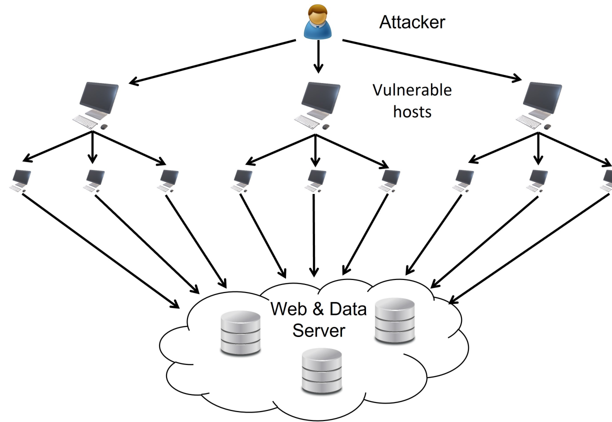


Figure 2.1: The high-level DDoS attack procedures

Fig. 2.1 illustrates a general DDoS attack model. Initially, a attacker scans and explores vulnerability in remote machines. Infected machines are referred to as zombies, which automatically and repeatedly infect more agents as new zombies and send attack packages to target victims. Finally, the attack prevents hosts from performing regular activities and addressing genuine received requests. The attack purposely makes hosts unavailable by flooding the resources of target systems with frequent requests [114].

- *Volume-based attacks.* This kind of attack aims to exhaust the bandwidth of a victim's site. When it comes to applications that involve interactions among multiple users, infection by other users is often necessary to warrant applications' desired functionality. The ability of an attacker to infect such applications is fundamental because these applications, containing shared data, are inherently constructed to be accessed by multiple users including collaborative users [94].

- *Protocol-based attacks.* This type of attack attempts to exhaust a server's versatile resources. For instance, a ping of death attack delivers an oversized ICMP datagram (encapsulated in IP packets) to a victim's computing node. The ping command fully utilizes an ICMP echo request and echo reply messages, which are commonly applied to check if a remote host is healthy and alive. In this case, a ping of death attack may cause the remote system to hang, reboot or even crash [131].

- *Application-layer-based attacks.* Because of the improved robustness and bandwidth of servers and networks developed in recent years, attackers are no longer inclined to throttle network bandwidth of victims' servers. Rather, the attackers are focusing on exhausting server resources like CPU cycles, database cycles, main memory or socket connections [14].

Not surprisingly, DDoS is a serious threat to cloud storage systems. Understanding attack strategies is the first line of defense against DDoS threats. In the context of cloud storage systems, DDoS attacks share the following two distinctive features [25].

- *Strategy 1.* Flooding attacks mainly rely on botnets to attack target hosts or networks, forming a many-to-one attack modality to expand attack scope and to worsen the harm of attacks in a cascading manner. Each attack has a focal point where certain services become unavailable in a target network.

- *Strategy 2.* Given an open shared-resource platform lacking source IP address authentication or authentication capability, attackers make use of packet source IP spoofing to strike victims. Because regular traffic at a monitoring point ought to respond to destination and destination-to-source addresses, source IP addresses cannot receive a valid reply from the destination IP address.

  We depict the main stream DDoS attacks in Fig. 2.2. In application-bug level attacks, attackers exploit system vulnerabilities and weaknesses to render cloud resources unavailable for users. Among the common attack vectors are protocol vulnerability, system weakness, outdated patches and misconfigurations. For example, vulnerabilities in protocols used by target applications can be explored by hackers, who aim to crash applications by delivering specially crafted packets to overload the applications. [110]

  Infrastructure attacks is one of the challenging attacks to detect in the cloud and it gained greater attention with the research community [30]. the attackers only need the IP address of the target without the need to exploit any vulnerability. DDoS flooding

attack can be carried out in two different forms, namely a direct attack and a reflector attack.

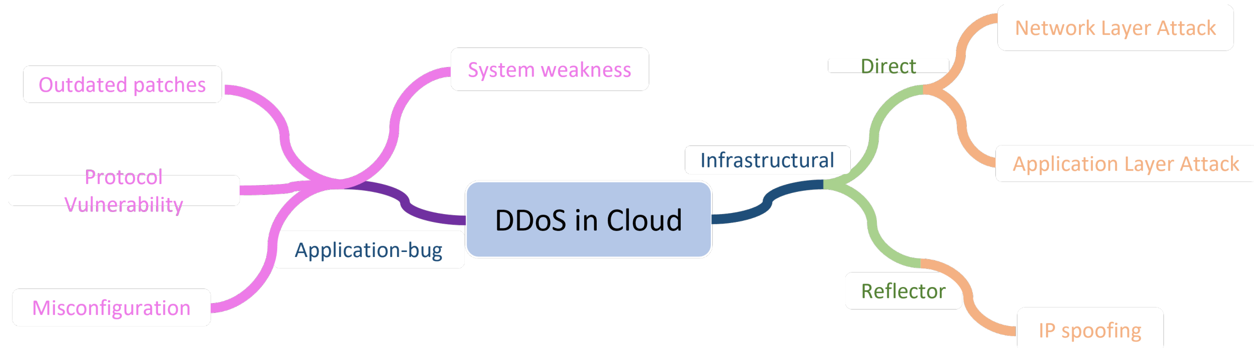


Figure 2.2: Main stream DDoS attacks in cloud computing platforms.

2.4.2 DDoS Detection Models in Clouds

To address the two strategies of DDoS threats in cloud storage systems, anomaly-based (see Section 2.4.3) and signature-based models were proposed to facilitate DDoS attack detection. The anomaly-based detection models utilize statistical methods for analyzing behaviors of data packets to sense any anomalies [57]. On the other hand, signature-based models intend to define a set of signatures or predefined knowledge from ground up, followed by identifying a candidate pattern as an intruder [76]. Apart from anomaly- or signature-based approach, a host of hybrid techniques were devised by mixing the two fundamental methods [21].

Existing signature-based techniques receive a wide adoption in detecting network-layer DDoS attacks. For example, *Snort* is one of the most commonly deployed signature-based detection techniques on multiple platforms [102]. Enlightened by the traditional signature-based approaches, Lee *et al.* implemented a data mining framework where inductively learned models is applied to build supervised classifiers [65]. This novel solution enables detection systems to discover anomalies and know intrusions by the virtue of mining consistent and useful patterns.

Reference	Detection technique	DDoS attack type
Virupakshar *et al.* [116]	Anomaly	Infrastructural
Kown *et al.* [64]	Anomaly	Not stated
Alqahtani *et al.* [9]	Anomaly	Infrastructural
Chen *et al.* [24]	Anomaly	Application-bug
Wang *et al.* [117]	Anomaly	Infrastructural
Meng *et al.* [80]	Anomaly	Infrastructural
Bakshi *et al.* [13]	Signature	Infrastructural
Karnwal *et al.* [59]	Signature	Application-bug
Gupta *et al.* [44]	Hybrid	Infrastructural
Modi *et al.* [83]	Hybrid	Not stated

Table II: The existing DDoS detection techniques

With regard to deployment positions, the signature- and anomaly-based detection mechanisms can be classified into application-level and network-level mechanisms. In the past decades, a majority of DDoS attacks were concentrated on network bandwidths because this resource type can be easily exhausted. With the advancement in robustness of network services, DDoS attacks against network bandwidth recently gave way to new attack modalities that exhaust server resources like CPU and memory at the application layer.

2.4.3 Application Layer DDoS Detection

Table II summarizes an array of DDoS Detection techniques for cloud storage systems. The prior studies were mainly focused on anomaly detection techniques that cope with infrastructural DDoS attacks. Therefore, we propose a novel anomaly detection solution in Section 5.2 after a thorough review of the existing anomaly detection schemes. From a technical point of view, DDoS attacks carried out at the application layer share the following distinctive characteristics [96].

- *Legitimate Requests.* An attack request mimic regular requests, making it arduous to discriminate attack requests from legitimate ones. Since the application layer DDoS

attacks proceed through legitimate HTTP packets, most of network-level filters and firewalls are unable to sense such anomaly requests from application attacks.

- *Limited Resources.* The bottleneck in application-layer DDoS attacks lies in server resources. With the innumerable number of requests, the Internet host can be exhausted sooner or later.

- *Targeted Attacks.* Application layer DDoS attack may occur on the various targets such as CPU, memory or socket connections. An attack can only attempt to exhaust one resource without affecting the other types of resources. Nevertheless, the entire system becomes unavailable under such an attack that deteriorates a single resource.

A raft of DDoS detection models are aiming to protect cloud storage systems from imminent threats of application layer DDoS attacks. Recall that attack requests and legitimate requests are alike, it is futile if not impossible to detect such attacks by simply examining individual requests. More often than not, the existing solutions are generally reliant on an analysis of requests stream tracking or requests template matching. For example, Ranjan *et al.* piloted one of the earliest studies focusing on the application layer, where an attack detection mechanism inspects characteristics of HTTP sessions by using statistical and rate-limiting [99]. Xie *et al.* proposed a popularity-aware techniques to detect DDoS for popular website through an access stream analysis [126]. On the flip side, SOAP-based DDoS repeatedly delivers SOAP requests to the URL, aiming to misuse the flexibility of security specifications. Rahaman *et al.* devised a detection model to specify stringent requirements, which should be fulfilled by incoming SOAP requests [98].

An increasing number of application-layer DDoS detection models are anchored on modern machine learning algorithms. For instance, the k-means clustering algorithm is adopted to classify incoming requests [129]. Meng *et al.* built a learning model powered by discrete-time Markov chains to compare users' current behaviors with normal history behaviors [80].

Umarani and Sharmila explored a naive bayes and k-nearest neighbor based classifier to identify if a request stream is legitimate through an access matrix from HTTP traces. Om *et al.* [62] integrated k-means, naive bayes and k-nearest neighbour to construct an anomaly detection approach, where a feature selection model removes irrelevant attributes to implement a clustering method.

The previous techniques were forged to maximize the prediction and detection accuracy of application-layer DDoS attacks. It is evident that detection accuracy and extra resource cost are somehow opposite aspects. Therefore, a handful of techniques were focused on both detection accuracy and overhead control. For example, Sreeram *et al.* assessed the feature metrics to diagnose request stream behaviors and to create a bio-inspired-anomaly-based model to cut back training and testing cost [111]. Jiang *et al.* developed a local measurement method accompanied by a remote counterpart to estimate the status of web servers, thereby achieving good trade-offs between the monitoring quality and overhead [55]. Doshi *et al.* restricted computational overhead by using a limited feature set in the variety of machine learning models [32].

Even though an array of existing techniques attempt to curtail detection overhead, high detection accuracy is normally obtained at the price of extra overhead. In this paper, we propose a similarity-analysis-based detection model catering for application-layer DDoS attacks. The overarching goal of our new model is to strike good balance between detection overhead and high accuracy.

2.5 Malware Detection in Cloud Storage Systems

Machine learning techniques are widely adopted in the malware detection field [4]. They generally fall into two camps, namely, static analysis [84] and dynamic analysis approaches [10]. In what follows, we articulate malware detection solutions from these two angles - static and dynamic malware detection. Fig. 2.3 summarizes the high-level procedures of static and dynamic malware detection schemes.

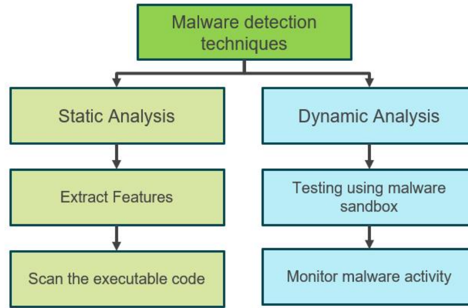


Figure 2.3: The high-level procedures of static and dynamic malware detection schemes.

Static Analysis Approaches. Amid a static analysis of malware detection, no execution of executable programs takes place. The goal of static-analysis-based approaches is to finish up malware detection tasks in a swift manner without being interrupted and slowed down by third-party programs. Such malware detection solutions entail the process of analyzing executable files by examining the code without executing them. A static analysis procedure is comprised of two steps. First, an executable file is disassembled or reverse engineering disassembled to retrieve the code. Next, detection of malware is carried out by scanning the executable code derived from the previous step. Smart malware may evade static-analysis-based methods by embedding syntactic code errors that mislead disassembly while performing functions during executions. Alternatively, an analysis can be accomplished by looking through executable binary files followed by applying machine-learning-based detectors to diagnose malicious software [12].

Deep learning models have been introduced into malware detection systems [12][105][106], where malware files are diagnosed without being executed. When it comes to online malware detection, most static analysis schemes found in the literature cope with single samples without addressing the mislabeling problem or time windows of identifying malicious patterns. [79]

Dynamic Analysis Approaches. Unlike static analysis counterparts, dynamic analysis approaches are normally deployed in an isolated environment such as sandboxes or virtual machines (VM). In such detection schemes, information is gathered during executions like

system calls, memory accesses, and network communications. The family of dynamic analysis solutions is applicable for malware-file classification in addition to online malware detection. It is noteworthy that online malware detection is closely related to intrusion detection systems [5].

Dahl *et al.* proposed a dynamic analysis solution to collect features from malware code that runs in a lightweight virtual machine [28]. Unfortunately, this approach is inadequate for online malware detection. Abdelsalam *et al.* devised an online malware detection system in clouds. This online detection system makes use of convolutional neural network (CNN) to maintain an optimal number of running virtual machines according to dynamic workload. More specifically, the number of virtual machines is dynamically scaled up or down based on load incurred malware detection. The online detection system is quite practical, because the system detects malicious behaviors while the other applications keep running on clouds. In a nutshell, machine-learning algorithms such neural network techniques are widely and judiciously employed to detect malware in clouds [29][4].

More often than not, hackers embark on large-scale distributed denial of service attacks or DDoS through malware code, phishing, and email spamming [119]. Many prior studies (see, for example, [128]) have addressed cloud security issues from various aspects such as networks, hypervisors, virtual machines, and operating systems, to name just a few. These cutting-edge security solutions are derived from rule-based intrusion detection systems accompanied by statistical anomaly detection models.

Chapter 3

A Popularity-Aware Data Reconstruction System

In this chapter, we describe the conceptual and mathematical underpinnings of the proposed POST system, which embraces a clustering module and a recommendation module. We start this section by introducing a well-known RS code deployed in POST. Next, we present a data clustering process during the data archival procedure where data are converted from *3X* replication into erasure-coded ones. We also shed some light on a user-based collaborative filtering algorithm, which generates a reconstruction queue aiming to shorten user waiting time during online data reconstruction.

The rest of this chapter is organized as follows. Section 3.1 describes the organization of our proposed popularity-aware data reconstruction model and the corresponding components, Reed Solomon-based EC, k-prototype clustering, and user-based collaborative filtering recommendation. The algorithm design of each component is proposed in Section 3.2. Section 3.3 gives a comprehensive system design of POST. Performance evaluation is presented in Section 3.4. This chapter's summary can be found in Section 3.5.

3.1 System Architecture

In this section, we describe the conceptual and mathematical underpinnings of the proposed POST system, which embraces a clustering module and a recommendation module. We start this section by introducing a well-known RS code deployed in POST. Next, we present a data clustering process during the data archival procedure where data are converted from *3X* replication into erasure-coded ones. We also shed some light on a user-based collaborative filtering algorithm, which generates a reconstruction queue aiming to shorten user waiting time during online data reconstruction.

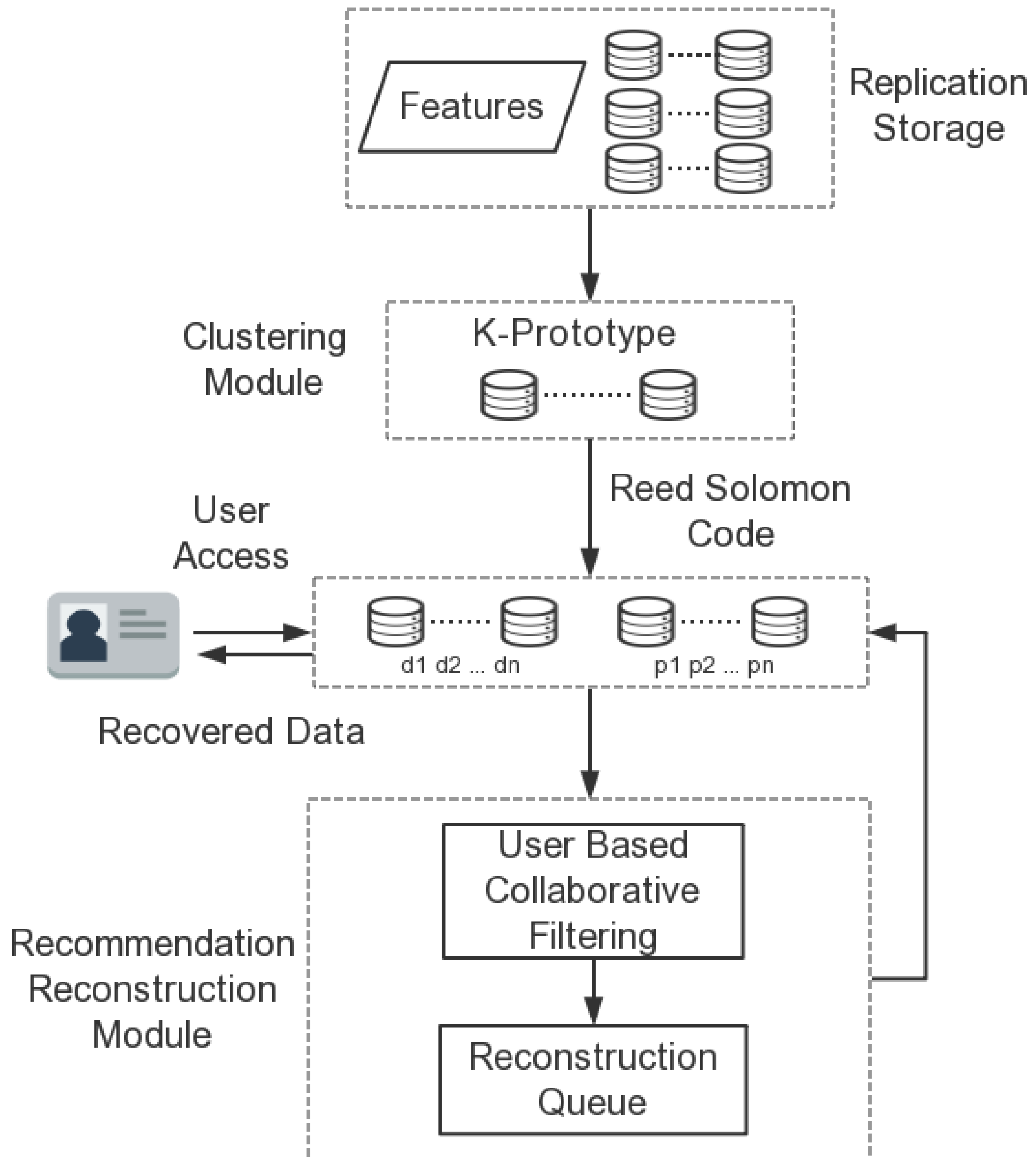


Figure 3.1: The architecture of POST, when warm data with features comes from 3X replication storage system, the Clustering part will archive data to erasure code based storage system by k-prototype algorithm. When users are accessing data with data block failures, recommendation part offers a recommendation list based on users set, and give a reconstruction list for each stripe in the storage system.

3.1.1 Overview

Fig. 3.1 depicts the system architecture of *POST*, which is a multifaceted data management system for storage clusters. First, POST archives warm data in erasure-coded storage systems. Second, POST is responsible for reconstructing data for faulty nodes. During the online data reconstruction procedure, POST makes judicious decisions on reconstruction order, in which popular data are assigned high priorities to reduce user response time.

POST embraces two subsystems, namely, the online data archival subsystem and the data reconstruction subsystem. We conducted an empirical study demonstrating that the data archival subsystem is conducive to booting the performance of the data reconstruction subsystem. For simplicity without loss of generality, we run the data archival subsystem once to compare the data reconstruction performance before and after data archival procedure. There are various ways to coordinate the data archival and reconstruction subsystems. For example, the data archival subsystem is invoked right before a data reconstruction process is initiated. Alternatively, one may embark on data reconstruction without kicking in the data archival subsystem, because the data archival subsystem is periodically managed and triggered by system administrators. Data archival intervals, of course, should be configured at the discretion of the system administrators. Apart of periodic executions, the data archival process might be running in full swing in a sporadic manner. Quantifying the impacts of data archival intervals on data reconstruction performance is beyond the scope of this chapter.

The POST system embraces the following three fundamental modules to manage data in an erasure-coded storage cluster.

- *The clustering module.* This module (see Section 3.1.3) classifies meta data in a way that similar data can be grouped together to form strips during the *RS* coding process.
- *The erasure coding module.* After the data clustering is accomplished, the erasure coding module (see Section 3.1.2) is kicked in to apply the Reed-Solomon code to store data blocks in stripes.

- *The recommendation module.* When users are accessing data in the erasure storage cluster, the recommendation module (see Section 3.1.4) generates recommendation lists to be merged into a single reconstruction list. Popular data are listed on top of the list, whereas non-popular ones are placed on the bottom. During the online data reconstruction, popular data are rebuilt in an earlier stage.

In the POST system, data are distributed across multiple nodes, the health of which is periodically gauged by sensors in the system (e.g., watchdog). A node becomes a faulty one when one sensor or more frequently reports erroneous measurements. It is straightforward to incorporate prevalent faulty-node techniques into POST. For example, [103] and [73] can be applied to detect node failures in storage systems. In POST, we offload the node monitoring and faulty-node detection functionality to the existing detection techniques.

We pay attention on optimizing data reconstruction performance in realm of single-node failures, because single-node failures are the most common case that ought to be efficiently handled. Importantly, POST has the capability to recover data from multiple faulty nodes, provided that multiple parity nodes are configured in erasure coded storage clusters. The number of faulty nodes tolerated in the storage clusters, of course, is reliant on the number of parity nodes. It is prudent to immediately kick in data reconstruction upon the detection of the first faulty node rather than postponing the recovery until subsequent faulty nodes emerge. In doing so, the overall reliability of storage clusters can be enhanced.

It is arguably true that POST is adroit at governing data reconstruction in the worst case where multiple nodes concurrently fail. Regardless of multi-faulty-node or single-faulty-node cases, POST prioritizes reconstruction requests in accordance to gauged data popularity. Compared with the single-node recovery scenario, the multi-faulty-node case takes a longer time period to reconstruct data.

3.1.2 Erasure-Coded Storage Clusters

Erasure codes effectively reduce storage cost by using redundancy blocks. In this part of the dissertation study, we use Reed-Solomon coding or RS to build an erasure-coded storage cluster; RS is a systematic code where the encoding process is a simple application of linear algebra. If a storage cluster adopts RS coding of the structure $(k+r, k)$, the storage system has k *data blocks* and r *parity blocks*. In an erasure-coded storage system, data are encoded to construct parity blocks by multiplying k data blocks with a k×(k+r) *generator matrix*, which involves a k×k *identity matrix* and a k×r *redundancy matrix* (see, for example, Fig. 3.2) [77]. In practice, a Vandermonde matrix is applied to encode RS codes [100][92], the coefficient $\alpha_{0,i}$ is 1, where i∈{0,1,...,k-1}.

$$r\left\{\overbrace{\begin{bmatrix} a_{0,0} & \cdots & a_{0,k} \\ \vdots & \ddots & \vdots \\ a_{r,0} & \cdots & a_{r,k} \end{bmatrix}}^{k}\right. \times \begin{bmatrix} D_0 \\ \vdots \\ D_k \end{bmatrix} = \begin{bmatrix} P_0 \\ \vdots \\ P_r \end{bmatrix}$$

Figure 3.2: Generating parity blocks in (k+r,k) Reed-Solomon codes.

A $(k+r, k)$ RS-coded storage cluster is comprised of an array of k+r storage nodes. k data blocks and r parity blocks are exclusively stored on k data nodes and r parity nodes in the storage cluster. Fig. 3.2 plots a $(k+r, k)$ RS-coded storage cluster, in which all the nodes (i.e., clients, manager, and storage nodes) are linked together through a switch.

To achieve high I/O throughput in the $(k+r, k)$ RS-coded storage cluster, a large data is encoded into k+r blocks distributed across k+r storage nodes. A block is partitioned into multiple stripes [93]. A collection of k data strips and r associated parity strips are referred to as a stripe [89]. Fig. 3.3 demonstrates the layout of a conventional RS-coded storage cluster.

In this chapter, we treat each stripe in a storage system as one basic unit, in which data blocks share similarities according to given features. Similar data blocks are placed into a single stripe by the k-prototype clustering algorithm (see details in Section 2.2). Put another

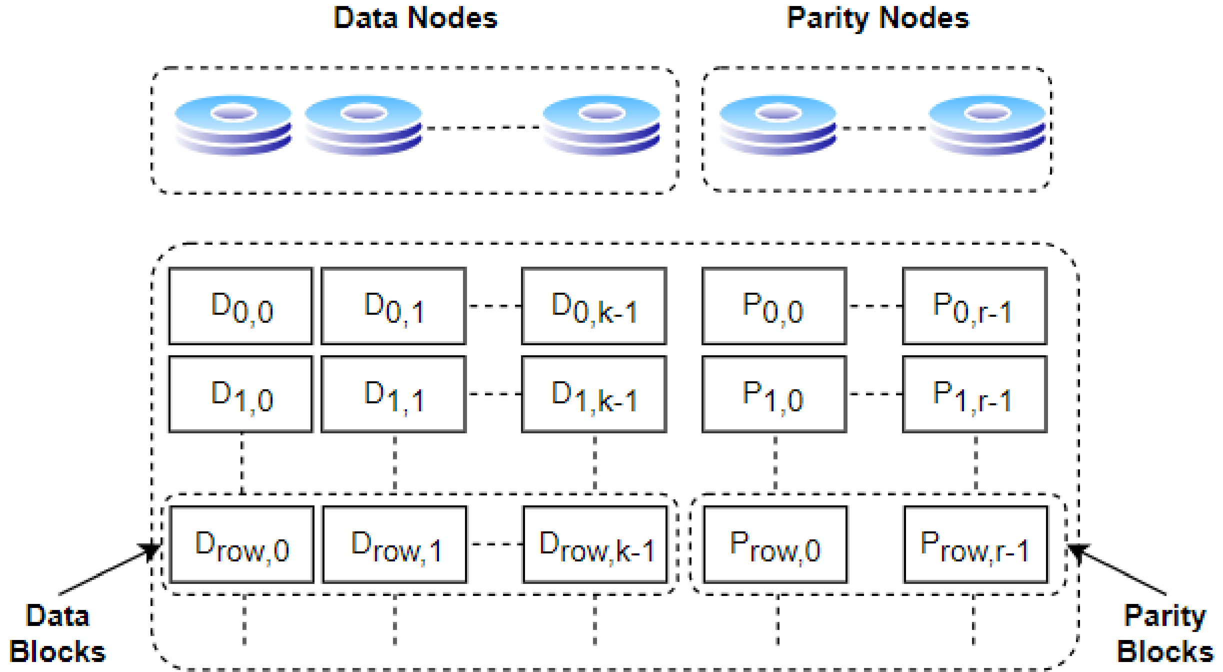


Figure 3.3: The layout in a conventional RS-coded storage cluster.

way, after the k-prototype algorithm performs data clustering, each stripe stores similar data blocks.

3.1.3 The k-prototype Module

From the perspective of users, data stored in a storage system have semantics to be exploited during the course of performance optimization. For example, movie data are comprised of genres, popularity, producers, release dates, and languages; common features of news articles include countries, types, and publishers. Our *POST* system incorporates the *k-prototype* clustering algorithm (see the clustering module in Fig. 3.1) to (1) group data with similar features and (2) place similar data within one stripe on a storage cluster during the data archival procedure.

The k-prototype algorithm is one of the most prominent algorithms to facilitate data clustering [54]. Given a large dataset, a small k value make the k-prototype approach

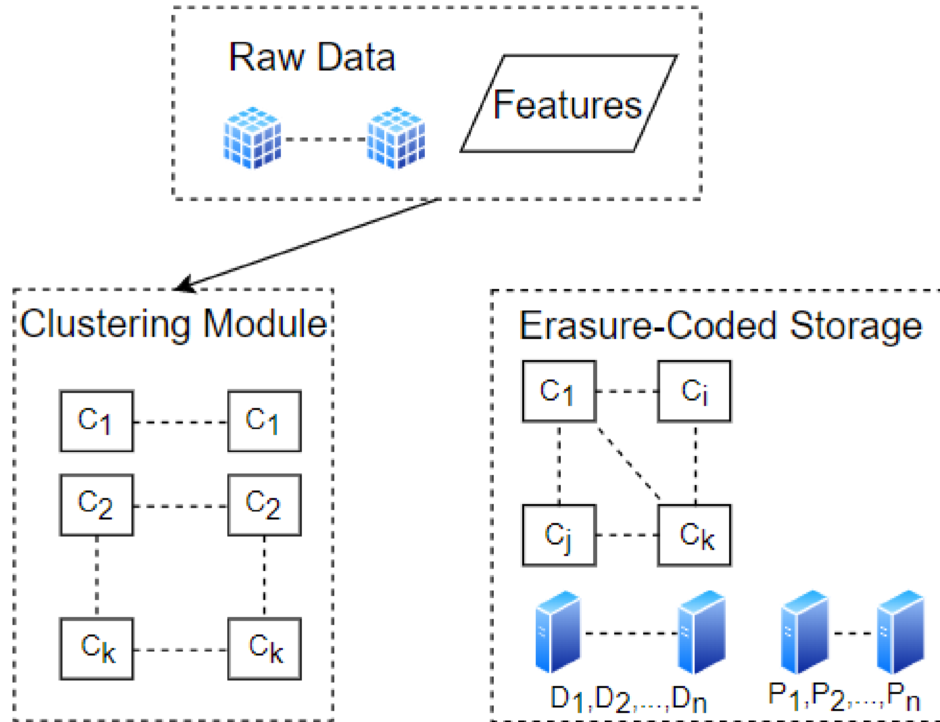


Figure 3.4: The work flow of k-prototype module.

outperform conventional hierarchical clustering techniques. In addition, k-prototype – a convergence guaranteed algorithm – is conducive to processing big data.

To optimize storage-system performance, system administrators advocate for the 3-way-replication scheme to store hot data (a.k.a., popular data) on storage clusters. When hot data are cooling down, non-popular data should be archived in an erasure-coded storage system to conserve storage space. The clustering module is triggered to classify data when the system embarks on the data archival process. More specifically, the clustering module plotted in Fig. 3.1 carries out the following four steps.

- To construct key-value pairs from data to be archived.
- To classify data according to their keys.
- To sort the key-value pairs according to the keys generated in the first step.

- To output classified key-value pairs (see details below) to the archival storage system, which is responsible for erasure coding.

We adopt the concept of key-value store into POST because key-value store is a data storage paradigm designed specifically for storing, retrieving, and managing associative arrays. Data items are organized in the format of key-value pairs, where the key represents an unique identifier (e.g., hash) and value is data content (e.g., images and documents). Generally speaking, the value of a key-value pair is stored as a blob requiring no upfront data modeling or schema definition. In our POST, we set key as a file's identifier (a.k.a., ID), which is referring to its data content - the value of its key-value pair. It is worth mentioning that values, stored in an erasure-coded storage system, are commonly gigantic in size in big data applications.

In the process of archiving warm data, meta data with features are delivered to the managing node in an erasure-coded storage cluster. The manage node invokes the clustering module (see Fig. 3.1 in Section 3.1.1) to classify data (i.e., key-value pairs) according to features of the data. The *k-prototype* algorithm is implemented in the clustering module, the output of which is a sorted list of keys from the key-value pairs. As a final phase of the data archival process, data with strong similarities are stored within one strip in erasure-coded storage cluster, which is in charge of creating parity blocks according to the Reed-Solomon code.

3.1.4 The Popularity Calculator Module

Recall that the four popularity-driven systems designed in Chapter 3, Chapter 4, Chapter 5 and Chapter 6 share a common and vital module: a popularity calculator. The goal of the popularity calculator provides a popularity list for online big data systems to reschedule the data processing sequence. This section details the design of the core and vital component applied throughout the entire dissertation study.

The collaborative filtering algorithm, widely used in recommendation systems, offers users personalized recommendation lists when the users log into the systems. User preferences are predicted, allowing the users to access their favorite data without searching for the entire storage. We adopt the *user-based collaborative filtering* algorithm or UBCF [133][47] in this dissertation; the rationale behind this choice is three-fold. First, UBCF benefits from historical access patterns of an enormous number of users. A large set of active users enable UBCF to make practical recommendations without relying on subject area expertise. Second, UBCF is flexible across a versatile of application domains. Collaborative filtering approaches are well suited to highly diverse sets of items (e.g., multimedia data). Third, UBCF tends to produce serendipitous recommendations, where accuracy might not be the highest priority. A majority of users have interests that span a wide range of subsets, which in theory can result in a diverse and interesting recommendations. The UBCF relies on the following three assumptions [133].

- People have similar preferences and interests.
- The preferences and interests are stable.
- A user's future choice can be predicted according to past preferences.

The collaborative filtering algorithm predicts people's preferences by (1) comparing of one user's access behavior with the historical behaviors of the other users and (2) finding the nearest neighbors to project the user's future interests or preferences.

Fig. 3.5) shows the work flow of the recommendation module. When users are accessing into the system, recommendation module originates recommendation list for all users, then combine them into one reconstruction list based on the popularity of data. If users request data object which is not available(a.k.a failed and has not been recovered), the data object will be inserted into the reconstruction list as the first one.

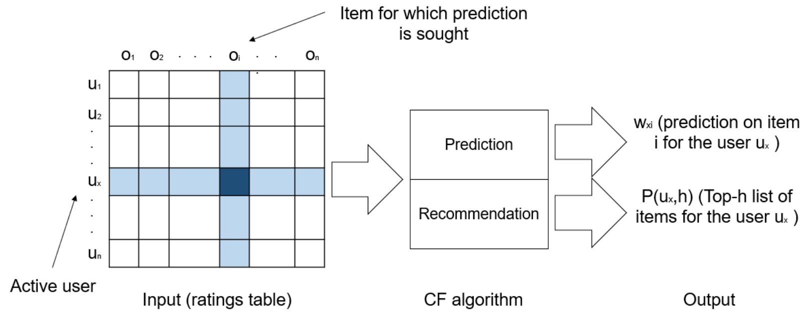


Figure 3.5: The work flow of recommendation module.

3.2 Designing Basic Modules

This section is centered around the two basic modules, namely, the clustering algorithm (see Section 3.2.1) and the filtering algorithm (see Section 3.2.2). Section 3.3 presents a way of seamlessly integrating the clustering and filtering algorithms in the context of data reconstruction in erasure-coded storage systems. To facilitate the presentations of the algorithms, we summarize the symbols and notation used throughout this manuscript in Table I.

3.2.1 k-prototype Clustering

In the data archival process, keys in key-value pairs are classified by the *k-prototype* algorithm. Data clustering plays a vital role, because similar data (i.e., values in key-value pairs) are stored within one stripe in an erasure-coded storage system. The *k-means* algorithm, a probabilistic solution, earns its reputation from high efficiency in clustering large data sets. In the k-means algorithm, each data item is mapped to the *Euclidean space*, where distances among data points can be measured.

The overarching goal of the POST's clustering module (see Fig. 3.1) is to group neighbouring data points together in one cluster, making a long distance between any pair of two clusters. To achieve this goal, we implement the k-prototype algorithm, which extends the conventional K-means algorithm by processing categorical data in addition to numeric ones [51].

Table I: Symbols and Notation

Symbol	Annotation
O	a dataset including a set of data objects
C	a set of clusters grouping data objects in O
N	the number of numeric features in data objects
N'	the number of categorical features in data objects
S	a set of center points for clusters in C
o_i	the ith data object in dataset O
e_i	key of the key-value pair in ith data object
v_i	value of the key-value pair in ith data object
c_k	the kth cluster yielded by k-prototype
s_k	a center point in cluster $c_k \in C$
$f_{i,j}$	jth feature data in the ith data object
$d(o_i, s_k)$	*Euclidean distance* between object o_i and center point s_k
$d(o_i, S)$	a set of distances between object o_i and all center points $s_k \in S$
$d(o_i, s_g)$	the minimal distance in distance set $d(o_i, S)$
θ	weight parameter for categorical features
$sim(u_x, u_y)$	the similarity value between users x and y
R_{xy}	a set of data objects rated by both user x and user y.
$r_{xi} \quad r_{yi}$	rate value of user x or y on data object i
$\bar{r}_x$	the average ratings of user u_x
H	a set of similar neighbours of a certain user x
q	the number of users in the system
q'	the number of similar neighbours of a certain user in the system
W	a set of predicted ratings value
$w_{x,i}$	the predicted rating of data object o_i for user u_x
$p(u_x, o_i)$	the predicted weight $w_{x,i}$ of data object o_i for user u_x
$\phi_{x,i}$	an object-weight pair records the data object o_i and the corresponding weight $w_{x,i}$
$P(u_x, h)$	records h highest recommended data objects o_i and corresponding weight in $p(u_x, o_i)$ for u_x
B	a set of faulty stripes in erasure-coded storage system
b_k	kth failure stripe $\in B$

Let $O = \{o_1, o_2, ..., o_n\}$ be a set of n data objects in a dataset, where objects are organized in the format of key-value pairs. The ith key-value pair is modeled as $o_i = (e_i, v_i)$, where e_i and v_i represent the key and value of data object o_i. The goal of the clustering module is to partition dataset O into m clusters, which is referred to as set $C = \{c_1, c_2, ..., c_m\}$. It is noteworthy that the data objects are grouped in accordance to their keys rather than values. More specifically, all the keys of data objects in cluster c_i are similar to one another. Similarities among data objects are gauged in terms of distances among keys (see Eq. 3.1 for details). Each key is comprised multiple features, which are mapped in an *N-dimensional* coordinate system (i.e., N is the number of features). Thus, key e_i can be expressed as a vector $e_i = (f_{i1}, f_{i2}, ..., f_{iN})$.

The key management in POST orchestrates keys in key-value pairs in a centralized manner. The core data structure is a tree-like concatenation of b-trees, in which each layer - dedicated to a fixed length of keys - effectively handles binary keys with arbitrary lengths.

Aiming to create a total of m clusters, the k-prototype algorithm randomly originates m points as centerpoints, each of which is reserved for a cluster. Let $S = \{s_1, s_2, ..., s_m\}$ be a set of centerpoints. Ideally, the m centerpoints should be far away from one another in the coordinate system. This requirement is likely to be fulfilled by random centerpoint selections. In case two randomly chosen centerpoints are too close to each other, the two centerpoints will be updated in a way that the two points are separated with a far distance (see Eq. 3.4 for details).

Next, the algorithm computes the distance between a candidate data object o_i with all the centerpoints. In what follows, let us define the distance between a data object (e.g., o_i) and a centerpoint (e.g., s_k in cluster c_k). Because the distance is a measure of similarity between any pair of two data objects, we make use of *Euclidean distance* to calculate the distance between data object o_i and centerpoint s_k. Thus, we have

$$d(o_i, s_k) = \sqrt{\sum_{j=1}^{N}(f_{ij} - f_{kj})^2}, o_i \in O, s_k \in S, j \in [1, N], \tag{3.1}$$

where o_i is a data object and s_k is a centerpoint in cluster c_k, f_{ij} and f_{kj} are the jth feature data of object o_i and centerpoint s_k. Distance measure $d(o_i, s_k)$ incorporates all the N features in the two data points.

We introduce distance set $d(o_i, S)$ to include the distances between data object o_i and all candidate centerpoints available in set S. Hence, we have

$$d(o_i, S) = \{d(o_i, s_1), d(o_i, s_2), ..., d(o_i, s_N), \} \tag{3.2}$$

the k-prototype algorithm places object o_i into cluster c_g so that distance $d(o_i, s_g)$ is the minimal item in distance set $d(o_i, S)$. In other words, comparing the distances between data object o_i and all the centerpoints in S, the distance between o_i and centerpiece s_g is the shortest one. Thus, we can express such a shortest distance $d(o_i, s_g)$ as

$$d(o_i, s_g) = \min_{\forall s_j \in S} d(o_i, s_j). \tag{3.3}$$

Center point s_k ($s_k \in S$) should be updated by calculating the mean position of all the data objects in cluster c_k ($c_k \in C$). Thus, the update center point s_k is written as

$$s_k = \frac{1}{|c_k|} \sum_{o_i \in c_k} o_i. \tag{3.4}$$

where c_k is the kth cluster, $|c_k|$ denotes the number of objects in cluster c_k, the summation of c_k means the summation of all the objects in cluster c_k with respect to all the features. Therefore, center points_k is a vector including the number N of features. Importantly, all the center points (i.e., $s_1, s_2, ..., s_m$) in set S are updated in the same manner prescribed in Eq. 3.4.

Finally, the algorithm repeatedly carries out the above steps (see Eqs.3.1-3.4) to group data in to the number m clusters. Generally speaking, the algorithm converges and partitions data objects into m clusters; nevertheless, there is no guarantee for a converged data-clustering result due to the problem's NP-hard complexity. The algorithm terminates under one of two conditions: (1) all the specified iterations are performed and (2) the center points can't be further updated. The bottom line is that the algorithm ensures to converge quickly to a local optimal solution.

The k-prototype algorithm is capable of handling categorical features. Let N' be a set of categorical features, the value of which cannot be quantified. To assess the distance between data object o_i and center point s_k of cluster c_k, we first introduce the distance $d'_j(o_i, s_k)$ between o_i and s_k from the perspective of the jth categorical feature. The distance $d'_j(o_i, s_k)$ is defined as a step function. If f_{ij} and f_{kj} are identical, distance $d'_j(o_i, s_k)$ is 0; otherwise, the distance is measured as 1. Thus, distance $d'_j(o_i, s_k)$ is expressed as

$$d'_j(o_i, s_k) = \begin{cases} 0, & \text{if } f_{ij} = f_{kj} \\ 1, & \text{otherwise.} \end{cases} \tag{3.5}$$

Now we derive the distance $d'(o_i, s_k)$ between categorical data o_i and center point s_k from Eq. 3.5 as a summation of the distances with respective of all the N' categorical features. Hence, we have

$$d'(o_i, s_k) = \sum_{j=1}^{N'} (d'_j(o_i, s_k)). \tag{3.6}$$

In case a dataset contains both numerical and categorical features, we combine Eq. 3.1 and Eq. 3.6 to measure the distance $d(o_i, s_k)$ between data o_i and center point s_k. More prosaically, distance $d(o_i, s_k)$ is a weighted summation of the distances derived from both

the numerical and categorical features. Hence, we have

$$\bar{d}(o_i, s_k) = d(o_i, s_k) \; + \; \theta \times d'(o_i, s_k)$$
$$= \sqrt{\sum_{j=1}^{N}(f_{ij} - f_{kj})^2} \; + \; \theta \times \sum_{j=1}^{N'}(d'_j(o_i, s_k)). \quad (3.7)$$

where θ is a weight parameter to adjust the importance of categorical features. After all the distances are quantitatively measured, each data object in the dataset is grouped to the nearest cluster.

3.2.2 User-Based Collaborative Filtering

During the course of online data reconstruction, data stored on faulty nodes are rebuilt while users are accessing data in the system. To shorten user I/O response time, the collaborative filtering algorithm generates a recommendation list, in which popular data that are likely to be referenced will be reconstructed first.

The collaborative filtering algorithm deployed in our POST system recommends data objects (a.k.a., data contents) to active users by comparing ratings of similar users. It is arguably true that recommended objects are more likely to be accessed by users and; therefore, POST leverages the recommendation list coupled with user-request lists as a guideline to reconstruct data. In doing so, data with a strong access likelihood will be given a high priority during the data reconstruction process.

The collaborative filtering module in POST is responsible for creating a recommendation list by computing a similarity value between any pair of two users in terms of their content ratings (a.k.a., user-item ratings). There are two steps in generating the recommendation list:

- To find a user set in which users share similar interests to a active user.
- To project the active user's interested data that the user has never accessed before.

Our system manages user-item ratings as user profiles, which are organized in the form of a user-item rating matrix. A rating matrix consists of a table, where each row and each column represent a user and a data object, respectively. The value sitting at the intersection of a row and a column denotes a user rating.

The collaborative filtering algorithm calculates the similarities between users with respect to user-item ratings, thereby making recommendations to active users in accordance with the observed behaviors of similar users. Similarities can be measured in various approaches like *Cosine*, *Pearson*, and *Euclidean* [133]. In our POST implementation, we choose to apply the *Pearson correlation coefficient* algorithm to compute similarities between any pair of two users $u_x \in U$ and $u_y \in U$, where U is a user set. Hence, the similarity between u_x and u_y is formally expressed as

$$sim(u_x, u_y) = \frac{\sum_{i \in R_{xy}} (r_{xi} - \bar{r}_x)(r_{yi} - \bar{r}_y)}{\sqrt{\sum_{i \in R_{xy}} (r_{xi} - \bar{r}_x)^2} \sqrt{\sum_{i \in R_{xy}} (r_{yi} - \bar{r}_y)^2}}. \tag{3.8}$$

where $\bar{r}_x$ and $\bar{r}_y$ are the average ratings from users u_x and u_y, r_{xi} and r_{yi} are ratings on data object o_i recorded by users x and y. R_{xy} is a set of data objects rated by both user x and user y.

The algorithm collects the q' nearest neighbours based on the similarity measures quantified by Eq. 3.8. Thus, the similarity set is written as

$$L = \{sim(u_x, u_1), sim(u_x, u_2), ..., sim(u_x, u_{q'})\}, q' < q. \tag{3.9}$$

where q' is the number of nearest neighbours of user x, q is the number of users in the system and $sim(u_x, u_i)$ is the similarity between the target user u_x and user u_x's neighbour u_y.

A predicted rating $p(u_x, o_i)$ is measured as a weighted average (see the second item on the right-hand side of Eq. 3.10) of neighbour's mean ratings plus active user u_x's mean rating $\bar{r}_x$. User biases become inevitable, because a handful of users tend to constantly give high or low ratings to all data objects. Thus, the predicted rating of user u_x on object o_i can be

expressed as

$$p(u_x, o_i) = \bar{r}_x + \frac{\sum_{y \in H}(r_{y,i} - \bar{r}_y) sim(u_x, u_y)}{\sum_{y \in H} sim(u_x, u_y)}, \tag{3.10}$$

where $sim(u_x, u_y)$ is the similarity between users u_x and u_y, $\bar{r}_x$ and $\bar{r}_y$ are embedded to alleviate user-associated biases. Throughout this manuscript, we also refer to the predicted rating $p(u_x, o_i)$ as weight $w_{x,i}$. Formally, we have

$$w_{x,i} = p(u_x, o_i). \tag{3.11}$$

A weight with a large value implies that data object i is popular. In other words, a high weight of an object indicates an excessive number of accesses to the object.

We introduce object-weight pairs as the fundamental data structure for recommendation lists. Given object o_i and its weight $w_{x,i}$ from with respect to user u_x, we express the object-weight pair $\phi_{x,i}$ as

$$\phi_{x,i} =< o_i, w_{x,i} > . \tag{3.12}$$

Given a user u_x, we denote $P(u_x)$ as a recommendation list with respect to u_x. Applying Eq. 3.10, we obtain all the predicted ratings (a.k.a., weights) of user u_x on all the data objects $\{o_1, o_2, ..., o_n\}$. Thus, we express the set of predicted ratings or weights with respect to user u_x as $\{\phi_{x,1}, \phi_{x,2}, ..., \phi_{x,m}\}$. We place these predicted ratings in set $P(u_x)$ in an non-increasing order. More formally, u_x's sorted predicted-rating set $P(u_x)$ is written as

$$P(u_x) = \{\phi_{x,1'}, \phi_{x,2'}, ..., \phi_{x,m'}\}, \quad \text{where} \quad w_{x,1'} \geq w_{x,2'} \geq ... \geq w_{x,m'}. \tag{3.13}$$

Now, we are in a position to form user u_x's recommendation list $P(u_x, h)$, which contains the top h ratings in the sorted predicted-rating set $P(u_x)$. Thus, recommendation list

$P(u_x, h)$ can be expressed as

$$P(u_x, h) = \{\phi_{x,1'}, \phi_{x,2'}, ..., \phi_{x,h'})\}. \tag{3.14}$$

To speed up the performance of the filtering algorithm, we implement this technique by the virtue of in-memory computing. Similar in-memory computing schemes tailored for big-data applications can be found in the literature (see, for example, [118])

3.3 Designing POST

We take a hierarchical design approach to the design and development of POST, in which a high-level controller is in charge of clustering (see Section 3.3.1) and another controller is responsible for data reconstruction (see Section 3.3.2). Both high-level controllers coordinate all the core modules in POST (see also the modules in Fig. 3.1 in Section (3.1.1)). In this subsection, we shed light on the algorithms for the k-prototype controller (see Algorithm 1) and the reconstruction controller (see Algorithm 2).

3.3.1 k-prototype Controller

The manager node governs the k-prototype scheme to calculate distance $d(o_i, s_k)$ (see Eq. (3.7)). Next, the distances between each data object and all the center points are computed, followed by grouping each data to its nearest cluster. In doing so, similar data objects are stored on the same strip in an erasure-coded storage system. Algorithm 1 shows the high-level controller that orchestrates (1) data clustering and (2) data archiving in POST.

Algorithm 1: The high-level controller orchestrates data clustering and data archiving.

Input:

Key-value pairs of data objects

Selected features of the data objects

Output:

1: storageQueue=$null$;

2: k-prototype($\{e_j\}, \{f_{jk}\}$);

3: **for** all clusters $c_i \in C$ **do**

4: **for** all data objects $o_j = (e_j, v_j) \in c_i$ **do**

5: storeData(e_j, v_j);

6: storageQueue.add(e_j);

7: **end for**

8: **end for**

9: **return**

In Algorithm 1, the k-prototype function takes a set $\{e_j\}$ of keys and a set $\{f_{jk}\}$ of features as inputs (see Step 2). The goal of k-prototype is to generate multiple groups (e.g., c_i), in which objects share similar features (e.g., f_{jk}) in their keys. All the clusters created by Step 2 are repeated processed by Steps 3-8. More specifically, e_j and v_j (see Step 4) indicate the key and value of the key-value pair for data object o_j; Function $storeData()$ is a process of storing the classified key-value pairs to data strips in the erasure-coded storage system in a sequential order (see Step 5). Step 6 stores all the values (e.g., v_j) of data objects as metadata into the erasure-coded storage system. In case users and applications intend to access data object o_j, the data can be referenced through its stored key e_j in the matadata manager.

3.3.2 Reconstruction Controller

The recommendation scheme offers a recommendation list (see also Eq. 3.14) for each user who is actively accessing the system. The reconstruction controller merges the recommendation lists of multiple users into a single data-reconstruction list (a.k.a., reconstruction list). Specifically, the controller carries out the four steps below to consolidate multiple recommendation lists into a reconstruction list:

- To retrieve data objects and their corresponding weights in user u_x's recommendation list $P(u_x, h)$ (see Eq. 3.14); to calculate the the number of occurrences and weight w_i for each key e_i.
- To map key e_i with stripes in the erasure-coded storage system.
- To calculate the summation of weights for each stripe, which is in faulty-stripe set B; to generate a reconstruction list containing a two-tuple of faulty stripe B and the corresponding weight W.
- To sort the reconstruction list by the decreasing values of weights of the stripes.

Algorithm 2 depicts the procedure of data reconstruction in POST, which carries out the above four steps to yield a single reconstruction list by merging all the recommendation lists for active users. The input information of Algorithm 2 include user I/O access history and ratings as well as $B = \{b_1, b_2, ..., b_{n'}\}$ - a set of faulty stripes in an erasure-coded storage system. The output is a reconstruction list *recList*, which contains stripe-weight pairs. The length of *recList* and the size of set B are identical. Thus, we have

$$recList.size() = |B| = n'. \tag{3.15}$$

Given stripe b_k and its weight w_k, we define $\alpha_k = (b_k, w_k)$ as a stripe-weight pair for the kth stripe. The reconstruction list - the output of Algorithm 2 - is formally expressed

as Eq. 3.16, where all the stripe-weight pairs are sorted by the Algorithm in the decreasing order of the weights.

$$recList = \{\alpha_1, \alpha_2, ..., \alpha_{n'}\} = \{(b_1, w_1), ..., (b_{n'}, w_{n'})\},$$
$$\text{where } w_1 \geq w_2 \geq ... \geq w_{n'}. \quad (3.16)$$

It is worth noting that the weight of a stripe resembles its popularity. The most popular stripes are listed at the head of *recList*; the least popular ones are placed at the end of *recList*.

Algorithm 2: The high-level controller of data reconstruction.

Input:

User I/O access history and rating records;

$B = \{b_1, b_2, ..., b_{n'}\}$; /* Stripes to be recovered */

Output:

$recList = \{\alpha_1, \alpha_2, ..., \alpha_{n'}\}$; /* A reconstruction list */

1: **for** all $u_x \in U$ **do**

2: $P(u_x) =$ UbasedCoFiltering(u_x); /* see Eqs. 3.10, 3.11 */

3: **end for**

4: **for** all $\alpha_k \in recList$ /* Initialize $recList$ */ **do**

5: SetWeight(α_k, 0); /* Initialize the weight of α_k */

6: **end for**

7: **for** all $u_x \in U'$ **do**

8: **for** all $\phi_{x,i} \in P(u_x)$ **do**

9: $o_i =$ GetDataObject($\phi_{x,i}$);

10: $w_{x,i} =$ GetWeight($\phi_{x,i}$);

11: $b_k =$ GetStripe(o_i, B);

12: recList[w_k]+= $w_{x,i}$;

13: **end for**

14: **end for**

15: Sort(recList[w_k]);

16: **return** $recList$;

In Algorithm 2, Steps 1-3 repeatedly perform user-based collaborative filtering to create recommendation lists for all the users in set U. Function $UbasedCoFiltering()$ in Step 2 implements the collaborative filtering strategy formally expressed in Eq. 3.14 Section 3.2.2.

We argue that the user-based collaborative filtering function should be executed in *Step 2* prior to building the recommendation lists. This order is reasonable, because regardless of data reconstruction, recommendation lists (see *Step 2*) are maintained and updated by modern storage systems. In a real-world setting, the recommendation lists are concurrently generated while users are accessing the system.

In Steps 4-6, all the weights of the stripe-weight pairs in $recList$ are initialized to 0. Let U' denote a set of users who are actively accessing the storage system during the data-reconstruction process. Since all users are maintained in user set U, U' is a subset of U (i.e., $U' \subseteq U$). *Steps 7-14* repeatedly calculates each user's weights with respective to data objects. More specifically, *Step 9* obtains the data object o_i from object-weight pair $\phi_{x,i}$ (see also the *GetDataObject*() function). *Steps 10* and *11* derive weight $w_{x,i}$ and stripe b_k from object o_i (see also the $GetWeight()$ and *GetStripe*() functions). In *Step 12*, weight w_k is updated by augmenting intermediate result $w_{x,i}$ obtained from *Step 10*.

Finally, the $sort()$ function in Step 11 sorts stripe-weight pairs in a non-increasing order of weights in reconstruction list $recList$.

In case there is no user request (i.e., $U' = \emptyset$), the system gracefully downgrades to offline data reconstruction where the reconstruction list is no longer required. In such an offline data-reconstruction case, *Steps 7-14* should be bypassed.

3.3.3 Time Complexity Analysis

The time complexity of user-based collaborative filtering implemented in *Steps 1-3* is $O(|U| \times |H|)$, where $|U|$ is the size of user set U and $|H|$ is the upper bound of the neighborhood size (see Eq. 3.10). There are two iterations (see *Steps 7* and *8*) in *Steps 7-14* and; therefore, the time complexity of creating weights (a.k.a., popularity) is $O(|U'| \times n)$, where

$|U'|$ is the size of user set U' and n is the number of objects. To be specific, we have the time complexity of creating weights, which is $O(n' \times \log(n'))$. The overall time complexity is

$$O(|U| \times |H| + n' \times \log(n') + n' \times n). \quad (3.17)$$

If the $UbasedCoFiltering()$ function can be pre-processed in an offline style, the time consumption should be mainly contributed by creating (see *Steps 7-14*) and sorting (see *Step 15*) a reconstruction list (i.e., *recList*). The time complexity can be simplified as

$$O(n' \times (\log(n') + n)). \quad (3.18)$$

3.3.4 Examples

We make use of a simple example to elaborate on the two controllers (see Sections 3.3.1 and 3.3.2) in POST. We start the example by focusing on k-prototype that archives data in an erasure-coded storage system. Fig. 3.6 shows the original data placement (see the layout on the left-hand side of Fig. 3.6) and the new data placement after clustering (see the layout on the right-hand side of Fig. 3.6). In this example, all the 24 data objects are grouped in four clusters, which are marked as A, B, C, and D.

After clustering is performed, the data objects are arranged in a way that, in most cases, data objects placed on one stripe belongs to the same cluster. For instance, objects $A1 - A4$ form the first stripe stored across the four disks (i.e., $disk1 - disk4$). If the number of objects in a data cluster exceeds the number of disks, multiple stripes should be originated for the cluster. For example, the size of cluster B is 7 (i.e., data objects $B1 - B7$); objects $B1 - B7$ are stored in two strips, of which the first one (i.e., $B1 - B4$) is archived on $disk1 - disk4$ and the second one (i.e., $B5 - B7$) is distributed on $disk1 - disk3$.

In this example all the data objects stored in a (4+4,4) RS-encoded system must be recovered due to the four faulty nodes. Suppose there are four users (i.e., $u_1, ..., u_4$), each of which issues two requests referred to as $r_{11}, r_{12}, r_{21}, ..., r_{42}$. Since *k-prototype* clusters data

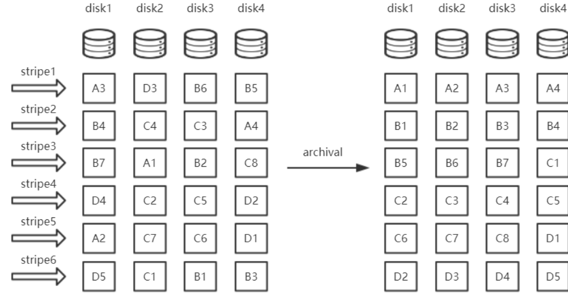


Figure 3.6: Data objects are arranged in a way that, in most cases, data objects placed on one stripe belongs to the same cluster. Objects $A1 - A4$ form the first stripe stored across $disk1 - disk4$. A large cluster may be handled by multiple stripes. Objects $B1 - B7$ are stored in two strips - ($B1 - B4$) and ($B5 - B7$), which are archived on $disk1 - disk4$ and $disk1 - disk3$, respectively. Objects $C1 - C8$ are placed among three stripes.

with similar features, users have a high probability to access data objects that belong to the same cluster. In this example, u_1 requests $A1$ and $A2$, u_2 accesses $B2$ and $B3$, u_3 requests $C3$ and $C4$, u_4 issues requests to $C4$ and $D5$.

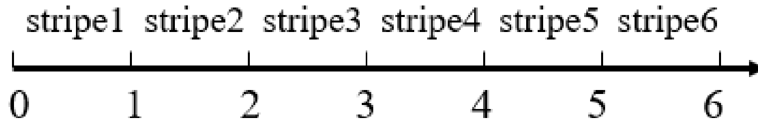


Figure 3.7: The case for k-prototype. The total waiting time as 13 with an average of 3.25 time unit per user.

Impacts of Data Clustering. Fig. 3.7 unravels the reconstruction time line for the six stripes shown in Fig. 3.6. For simplicity without loss of generality, each stripe can be recovered within a time unit. Intuitively, it takes six time units to cover all the stripes (i.e., $stripe1 - stripe6$.

We consider the user waiting times for a baseline case where there is no data clustering (see the layout on the left-hand side of Fig. 3.6). Because $A2$ is in $stripe5$ and $A1$ is in $Stripe3$, u_1 has to wait for the reconstruction of $stripe5$ before accessing all the requested data. Hence, user u_1 must wait for 5 time units before accessing all requested data. We calculate the waiting times for all the users in the similar manner. Thus, the total waiting time for all the four users to access the data objects in the baseline case is 19 time units with an average of 4.75 time unit per user. With data clustering in place (see the layout on

the right-hand side of Fig. 3.6), the total waiting time is shortened to 13 with an average of 3.25 time unit per user. Such an improvement is attributed by the fact that similar data objects are archived in the same stripes, which can be reconstructed together.

Impacts of a Recommendation List. Now we evaluate the sample where user-based collaborative filtering is involved without *k-prototype*. The data layout is illustrated on the left-hand side of Fig. 3.6. The reconstruction list is constructed using the weight values calculated from the recommendation lists, shown in Table III.

Table II: sample recommendation list

User	**recommendation**	**weight**
u_1	A1,A2	5, 4
u_2	B2,B3	5, 4
u_3	C3,C4	5, 4
u_4	C4,D5	5, 4

The four recommendation lists in Table II are combined into a single reconstruction list (i.e., *recList* in Algorithm 2). Fig. 3.8 depicts reconstruction behavior of the system governed by the reconstruction list.

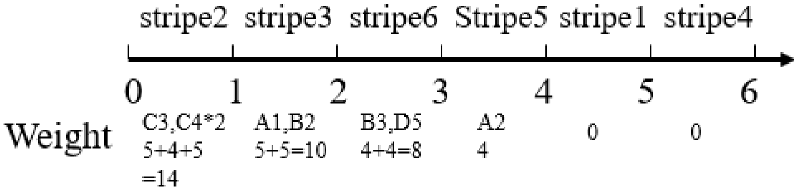


Figure 3.8: The case for recommendation list. The total waiting time as 11 with an average of 2.75 time unit per user.

Fig. 3.8 shows that the waiting time of user u_3 is 1 time unit, because u_3 can access all the requested data after *stripe*2 is recovered. As such, we obtain the total waiting time as 11 with an average of 2.75 time unit per user. This is the evidence that popularity-aware reconstruction can shorten user waiting time.

Impacts of Clustering and Recommendation. POST seamlessly integrates the clustering and recommendation strategies to optimize the reconstruction list. The impacts of clustering and recommendation are illustrated in Fig. 3.6 and Fig. 3.8), respectively.

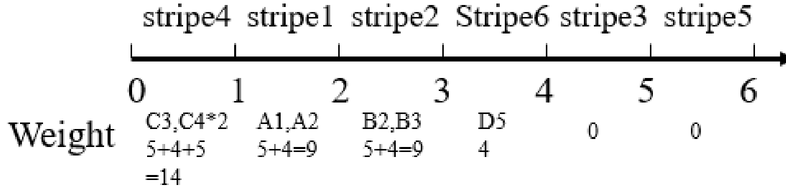


Figure 3.9: The case for POST. The total waiting time as 10 with an average of 2.5 time unit per user. The waiting times of the four users are 2, 3, 1, and 4, respectively.

Fig. 3.9 reveals the reconstruction outcomes of POST, where the total waiting time is as low as 10 time units with an average of 2.5 time unit per user. In the case of our POST, the waiting times of the four users (i.e., u_1-u_4) are 2, 3, 1, and 4, respectively. We conclude from this example that POST significantly shortens user waiting times during the online data reconstruction process.

Table III: sample recommendation list

	Total waiting time	**Avg. Waiting Time**	**Reduction**
The *greedy* system	19	4.75	0
The *clustering* system	13	3.25	31%
The *recommendation* system	11	2.75	42%
The *POST* system	10	2.5	47%

3.4 Performance Evaluation

In this section, we evaluate the performance of POST driven by real-world datasets. We compare POST with a baseline solution called *Greedy*, which represents data reconstruction policies that ignore k-prototype clustering and collaborative filtering. We measure the average user waiting times of data reconstruction in an erasure-coded storage system governed by both *POST* and *Greedy*. We also test *POST*'s two variants, which are referred to as *Clustering* and *Recommendation*. In the *Clustering* scheme, we enable the k-prototype clustering feature while disabling the collaborative filtering feature. In the *Recommendation* scheme, we turn on the collaborative filtering feature and turn off k-prototype clustering. All four strategies are assessed under a wide variety of workload conditions.

Before presenting experimental results, let us discuss the experimental settings in Section 3.4.1. The impacts of data similarity on the system performance are discussed in

Section 3.4.4. Sections 3.4.5 and 3.4.6 presents the impacts of the number of users and the number of user requests, respectively. Section 3.4.7 sheds some light on the effect of stripe capacity.

3.4.1 Experimental Settings

We quantitatively evaluate the performance of POST using a real-world dataset containing movie data objects [3]. The movie dataset records user ratings accompanied by users access history. More specifically, the dataset is comprised of 26 million ratings from a total of 270,000 users for all 45,000 movies. Fig. 3.10 illustrates the distribution of the movies with respect to the number of ratings.

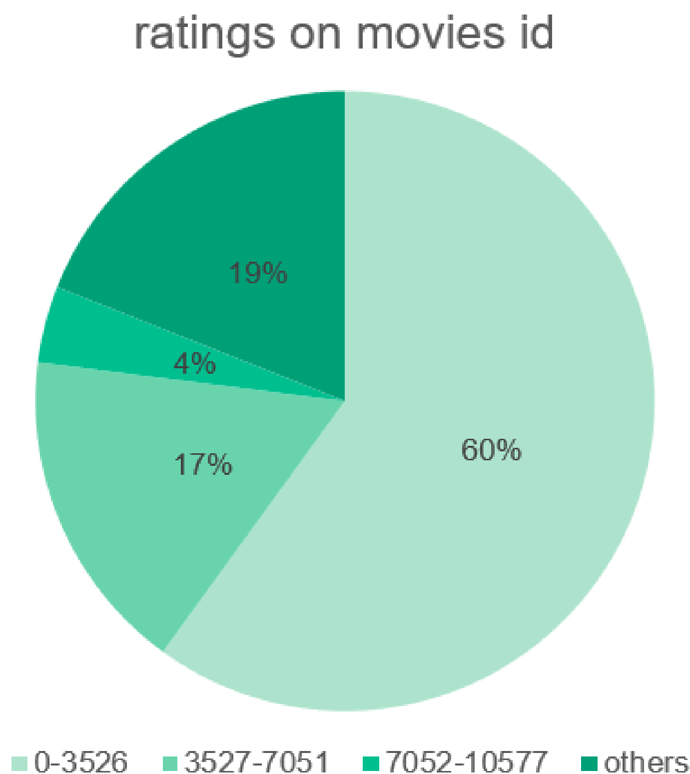


Figure 3.10: Distribution of user ratings with respect to *movie IDs*. Approximately 60% of ratings are placed on items, the *movie IDs* of which are smaller than 3,500.

We pay particular attention to this real-world dataset, because most movie data are classified as warm data where 80% data only capture less than 20% of accesses. The warm

Figure 3.11: distribution of user ratings for the movies dataset

data features of movie contents are confirmed by Fig. 3.11, which shows the distribution of user ratings with respect to *movie IDs*. Approximately 60% of ratings are placed on items, the *movie IDs* of which are smaller than $3,500$. As such, we select the *movie ID* ranging from 0 to $3,500$ in the dataset. All the ratings are on a scale of between 1 and 5 obtained from the official GroupLens website [3].

Importantly, movie data also have distinct numeric and categorical features (e.g., years, budgets, popularity, production companies, main actors, and genres), which are catalysts for the data clustering module (i.e., k-prototype clustering). The selected features in our implementation include *released year*, *runtime*, *genres*, *production companies* and *main actors*. In the k-prototype module (see Section 3.1.3), the number m of clusters is set to 10. In the recommendation module (see Section 3.1.4), weight parameter θ is configured to 0.8. To facilitate the test of average user waiting time for accesses to the data objects, the system concurrently serves various number (i.e., anywhere between 1,000 to 10,000) of data objects (a.k.a, movies) at the same time.

We also test our system driven by book data objects. Compared to the movie dataset, e-book dataset encompasses small objects; one stripe stores enormous book objects. The

book dataset contains ratings of ten thousand popular books. There are approximately 100 reviews for each book, the ratings of which varies one to five [2].

To carry out extensive experiments, we set up a cluster consisting of eight (8) storage nodes distributed as a (4+4,4) RS-coded storage cluster in addition to one manager node. In the stress tests of the data-reconstruction process, all the stripes in the system are recovered to replace four faulty nodes. To resemble real-world cluster computing systems (e.g., HDFS [109] and GFS [41]), we set data block to 64 MB and each stripe (i.e., $64MB \times 4$) can hold eight movie data objects or 64 book data objects.

3.4.2 Data archival

We measure time spent in archiving data, the process of which is following the k-prototype procedure. In POST, n data blocks of each data stripe are separated into k clusters. We investigate the impact of the number k of clusters on archiving performance by setting k to 5, 10, 15 and 20, respectively. Fig. 3.12 shows the data archiving time in POST when we configure the number of iterations, the number of data nodes, the number of parity nodes, data block size to 30, 4, 4, 64 MB, respectively. The archived dataset contains movie data objects of approximately 25 TB [3].

In Fig. 3.12, the blue line reveals the data archival time when the k-prototype algorithm is disabled (a.k.a., the baseline case); the four green bars represent the data archival time under various numbers of clusters. We observe from the experimental results plotted in Fig. 3.12 that data archival time measures in POST are almost identical to that in the baseline case. Compared to the long archival time consumption, the k-prototype clustering time merely accounts for a small proportion. It is evident that k-prototype clustering imposes marginal overhead in POST, suggesting that the overhead induced by the k-prototype algorithm becomes negligible when a sheer volume (e.g., 25 TB) of data is archived. Because the data archival process is a one time deal, the small cost of k-prototype clustering can be reasonably ignored.

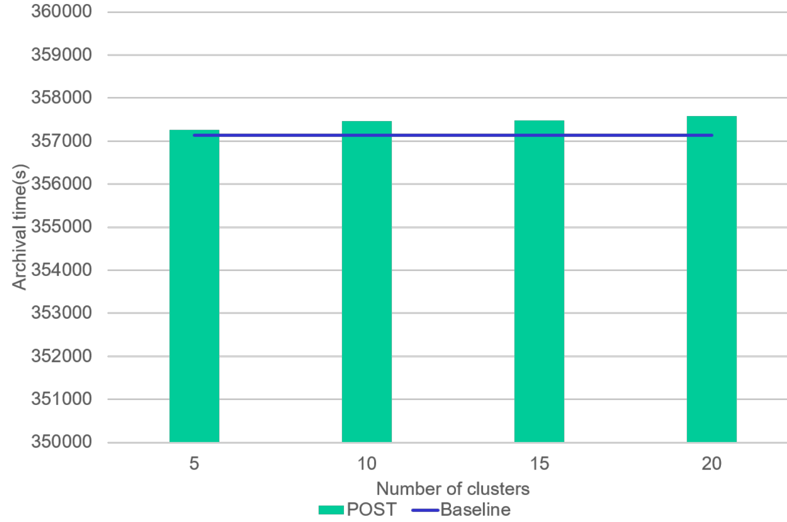


Figure 3.12: The impacts of the number of clusters on data archiving time in the POST system. The number of iterations, the number of data nodes, the number of parity nodes, data block size are configured to 30, 4, 4, 64 MB, respectively. The archived dataset contains movie data objects of approximately 25 TB.

3.4.3 Space Overhead Analysis

It is arguably true that POST's scheduling overhead in data reconstruction is quite low, because files are organized in form of large blocks to cut back meta-data management overhead. Meta data are implemented in POST to manage the popularity of data blocks. The meta-data manager keeps track of the popularity levels and tags of data objects stored in the POST system. It takes four bytes to hold the popularity and tag of each data object. Therefore, the meta-data management overhead in POST largely depends on the number of the data objects and data object size in a storage system. More broadly, the meta-data space overhead of each data object is as small as four bytes; the overhead ratio is quantified as a ratio between 4 bytes and the data object size (i.e., $\frac{1}{object-size}$). For example, when the data object size is set to 64 MB and the number of data objects is 1000, the space overhead of the meta data is 4 KB; the overhead ratio is 1/16M, which is approximately zero.

3.4.4 Impacts of Similarity

Recall that in this research we introduce a simple yet effective prediction module, where a recommendation list is continuously updated. These predicted results serve as a guideline to create a reconstruction list, which directly affects the performance of our POST. Fig. 3.13 depicts an overall impact of weight value $w_{x,i}$ on prediction accuracy. Intuitively, weight $w_{x,i}$ (see predicted rating in Eq. 3.11) is proportional to the prediction accuracy. The prediction accuracy dramatically deteriorates when the weight value is smaller than 3.

This result is vital and inspiring, because the correlation between the weight value and prediction accuracy offers a guideline to system administrators to optimize the h value in the recommendation list $P(u_x, h)$ (see Eq. 3.14 in Section 3.2.2). For example, when h is a large value, the predicted ratings placed at the end of the recommendation list (see Eq. 3.14) become inaccurate. On the other end of the spectrum, a small h value downgrades the performance of our POST.

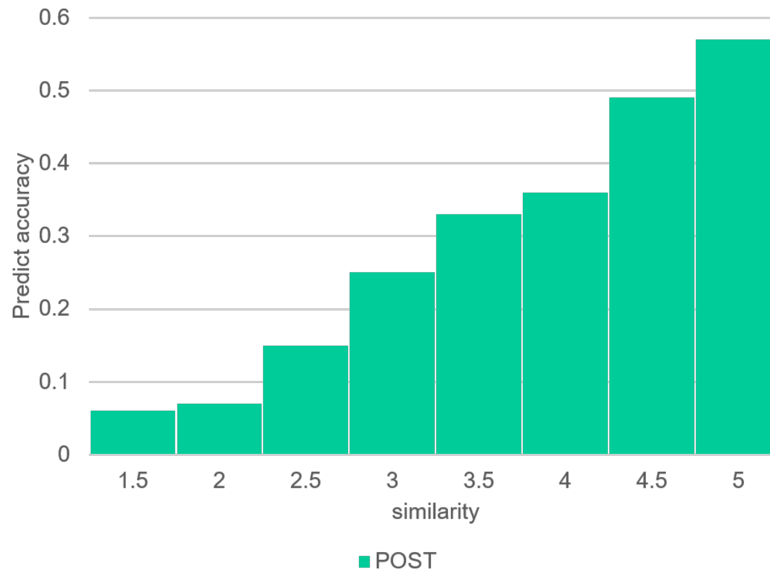


Figure 3.13: Impact of weight value on prediction accuracy. The weight value is proportional to the prediction accuracy. The correlation between weight value and prediction accuracy guides system administrators to optimize the h value in the recommendation list $P(u_x, h)$ (see Eq. 3.14.

POST's high-level controller strives to merge multiple recommendation lists to originate a single reconstruction list. We evaluate the time cost of high-level data-reconstruction controller (see also Steps 7−14 in Algorithm 2) by considering I/O overhead. The time cost is augmented to the total execution time of the POST system. Fig. 3.14 unveils the relationship between prediction accuracy and average user waiting time (i.e., waiting time). We normalize the waiting time of POST based on that of the greedy approach. The normalization results present the performance improvement of POST over the baseline greedy scheme. It is worth noting that the prediction accuracy is positive correlation with similarity value in k-nearest-neighbor(KNN) set L (see Eq. 3.9).

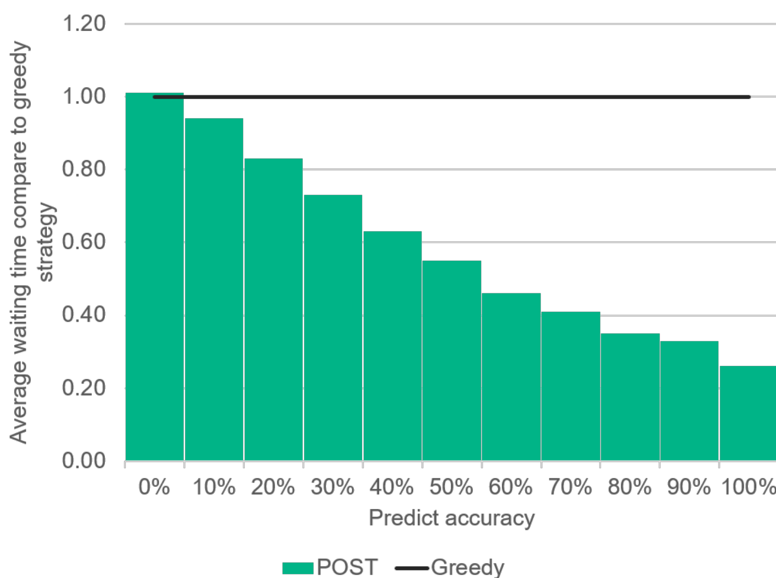


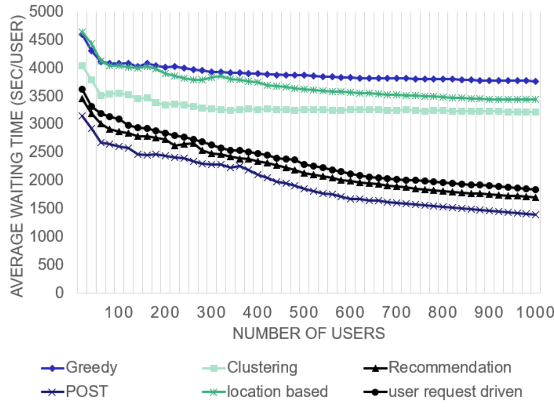
Figure 3.14: The impact of prediction accuracy on average waiting time. There are 100 users concurrently accessing 1,000 data objects. Boosting the prediction accuracy shortens waiting time.

Fig. 3.14 shows that boosting the prediction accuracy can significantly shorten the average waiting time. For instance, a prediction accuracy of 100% indicates the system is exactly aware of users' future interests, which lead to the most appropriate recommendation list. On the other hand, a prediction accuracy of 0% implies that POST is downgraded to the greedy scheme. This downgrading trend is reasonable, because the I/O overhead in Algorithm 2 is far lower than the reconstruction cost (about $\frac{1}{1000}$).

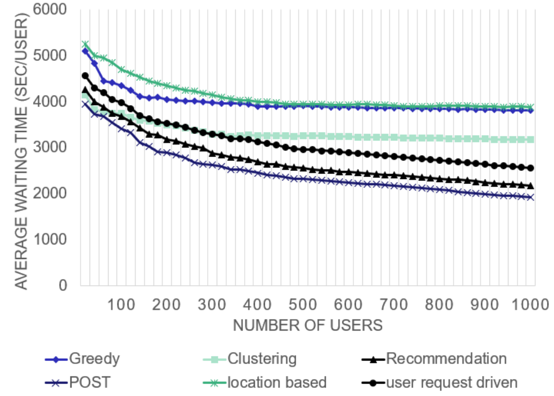
3.4.5 Impacts of Number of Users

In this group of experiments, we compare POST with two alternatives (i.e., the *clustering* approach, the *recommendation* approach) and three reconstruction strategies (i.e., the *Greedy* strategy, *location-based* strategy and *user-request driven* strategy). We quantitatively evaluate the effects of the number of users on the reconstruction performance of the six strategies. All the users issue their access requests, which are randomly selected from the users' history records. In particular, we randomly pick ten data objects from each user's history record to place on the access list. Fig. 3.15 unravels the sensitivity of our POST and the three counterparts to the number of users ranging from 20 to 1000 with an increment of 20.

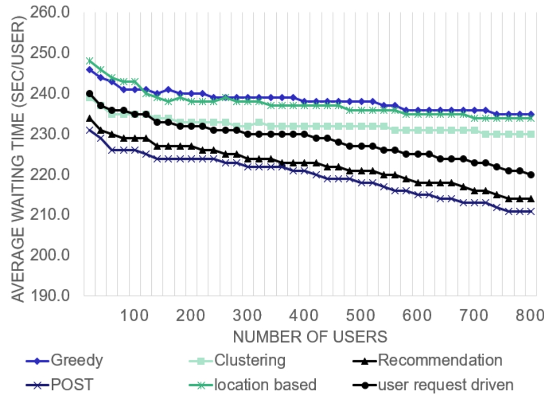
We randomly select ten (10) data objects from the access history of each user to perform as a request list. The average waiting time of a user is derived from the time at which POST finishes reconstructing the last movie in the user's request list.



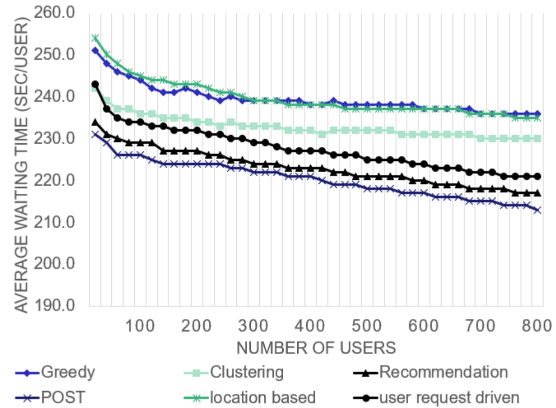
(a) The movie dataset of user group 1.



(b) The movie dataset of user group 2.



(c) The book dataset of user group 3.



(d) The book dataset of user group 4.

Figure 3.15: The average waiting time measures of the six reconstruction strategies managing the movie datasets and book datasets. The number of users varies from 20 to 1000 in movie dataset and 20 to 800 in book dataset.

Fig. 3.15 unveils that an increasing number of users in POST leads to a low average waiting time. For example, in Fig. 3.15(a), when the number of users is set to 100, the average waiting time in POST is 2571 seconds. In case of 400 users, the POST's average waiting time is reduced to 1858 seconds, representing a reduction of 28.7%. This trend is reasonable, because increasing the number of users makes it more likely to share movies among the active users. When the amount of shared data rises, an increasing number of users tends to benefit from reconstructing shared data objects. In other words, with the

increment of number of users, the probability of multiple users accessing the same data goes up accordingly. As such, the average waiting time is shortened thanks to requests sharing the popular data.

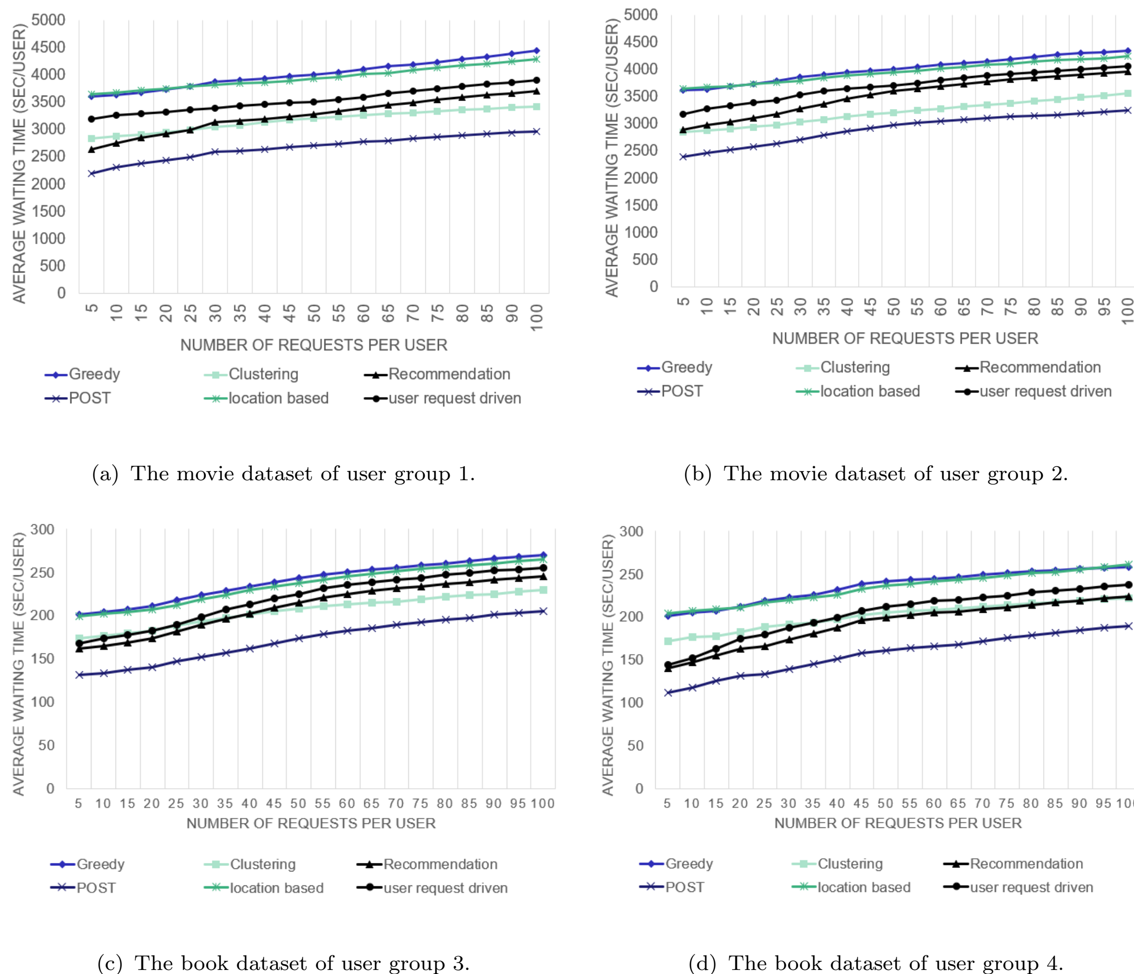
Not surprisingly, the average waiting time of the greedy scheme is insensitive to the number of users. Such a comparison between POST and Greedy demonstrates that when the recommendation algorithm is imported, popular data that have a high probability to be accessed by and shared among multiple users are ranked at the top of a reconstruction list. We conclude from this group of experiments that the POST exhibits superb and scalable reconstruction performance for a large number of users who tend to share popular data objects in storage systems.

We conduct an array of experiments to compare POST against the two common data reconstruction scheduling policies. Figs. 3.15(a) and 3.15(b) unveil the average waiting time of requests governed by POST, the *location-based* policy, and the *user-request driven* policy. We observe from Figs. 3.15(a) and 3.15(b) that the *location-based* data reconstruction scheduling policy behaves in a similar fashion as the *greedy* algorithm. It is evident that the scheduling decisions made by the location-based scheme are closely matching those of the *greedy* policy. More importantly, POST outperforms the *user-request driven* policy, in which the lack of popularity awareness makes it virtually impossible to prefetch popular data blocks.

Now we are positioned to compare Figs. 3.15(a) and 3.15(b), which show the experimental results of the movie datasets capturing the behaviors of two user groups (i.e., group 1 and group 2). We observe from Figs. 3.15(a) and 3.15(b) that all the tested algorithms perform better for user group 1 than user group 2. User group 1 enjoys more performance benefits than user group 2, because users in group 1 share more similar interests than those users in group 2. The results plotted in Figs. 3.15(c) and 3.15(d) are almost identical, because the interest similarity of users in group 3 is close to that of users in group 4.

3.4.6 Impacts of Number of User Requests

This experiment tests the performance affected by various number of requests per user. We fix the number of users to 100 while randomly choosing the number of requests per user from 5 to 100 with an increment of 5. Fig. 3.16 illustrates the impact of the number of requested data objects per user on average waiting time.



(a) The movie dataset of user group 1.

(b) The movie dataset of user group 2.

(c) The book dataset of user group 3.

(d) The book dataset of user group 4.

Figure 3.16: Average waiting times of the six reconstruction strategies. The number of requested data objects per user varies from 5 to 100 with an increment of 5. The number of users is set to 100.

We draw a total of four observations from Fig. 3.16. First, among the six competitive schemes, our POST is a front runner across all the settings. This experimental result is

consistent with that revealed in Fig. 3.15 in Section 3.4.5. Second, the average waiting time consistently increases with the increment of requests per user. This performance trend is applicable to all the evaluated six schemes.

Third, when the number of requests for each user is small, the *recommendation* system performs better than the *clustering* system. This observation suggests that the performance comparisons between the *recommendation* and *clustering* systems largely depend on the number of requests issued per user. The performance of the *location-based* system is similar to that of the *greedy* system, because the only denominating parameter affects the data reconstruction scheduling policy in the *location-based* and *greedy* systems is the data requests. Also, we observe that the overall performance of the *recommendation-based* system is superior to that of the *user-request driven* one. Thanks to popularity awareness, the *recommendation-based* system outperforms the *user-request driven* system.

Fourth, regardless the number of requests per user, the recommendation module in POST cuts back the average waiting time by approximately 20%. Such an improvement is insensitive to the number of requests per user, because the prediction accuracy is independent of the number of requests issued by a user. It is arguably true that in the *recommendation* system, increasing the number of requested data objects does not necessarily rank the popular data on the top of the reconstruction list.

Fifth, unlike the *recommendation* system, the *clustering* system consistently reduces the average waiting time when the number of requests per user goes up. The reason is that increasing the number of requests per user boosts the chance of placing the requested data in the same cluster, thereby making it more likely to store the data within the same stripe. We conclude from observations four and five that the recommendation system only offers an optimization at a fixed rate; the clustering system, on the other hand, can continue in slashing the average waiting time when we push up the number of requests per user.

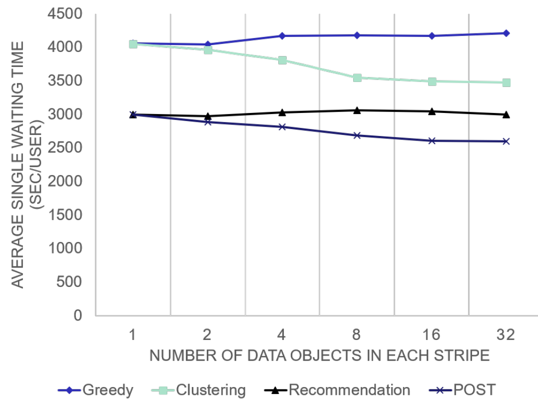
Last, but not least, the performance improvements of POST over Greedy become more pronounced when the number of requests per user is lifted. We attribute this trend to the

fact that when each user issues more requests, the requested data objects have a higher probability to be classified and stored in the same cluster; that is, these objects are more likely to be placed within the same stripe. In a nutshell, increasing the number of requests per user allows POST to deliver outstanding performance improvements over the Greedy scheme.

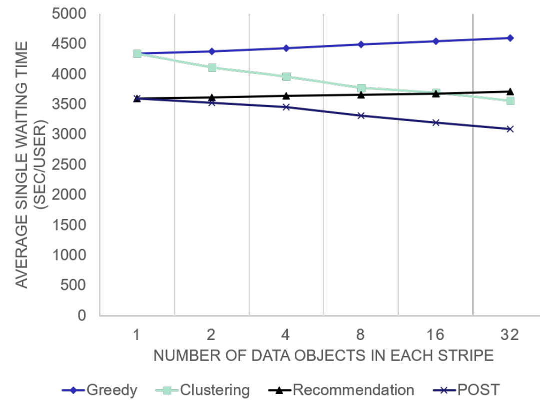
Comparing Figs. 3.16(a), 3.16(b), and 3.16(d), we observe that the clustering system outperforms the recommendation system when the number of requests per user goes up. In contrast, Fig. 3.16(c) illustrates that the recommendation system is superior to the clustering system. The results indicate that increasing the number of requests per user imposes a marginal impact on recommendation accuracy.

3.4.7 Impacts of Stripe Capacity

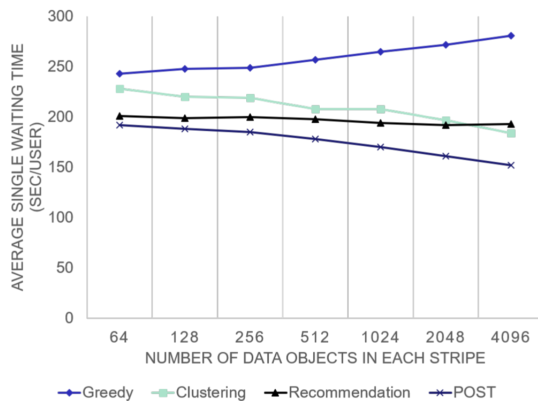
In the following experiments, we investigate the impacts of stripe capacity on POST's average waiting time. To achieve this goal, we configure the number of users to 100; we fix the number of requests per user to 10. Also, we set the data-block size from 8 MB to 2 GB, implying that each stripe is capable of holding anywhere between 1 to 32 movie data objects or between 64 and 2048 book data objects. Fig. 3.17 unravels the performance impacts of the stripe capacity on the four data reconstruction schemes. The stripe capacity is gauged in terms of the number of data objects stored in each stripe.



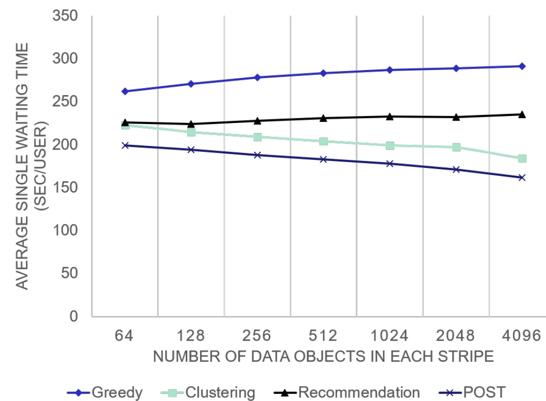
(a) The movie dataset of user group 1.



(b) The movie dataset of user group 2.



(c) The book dataset of user group 3.



(d) The book dataset of user group 4.

Figure 3.17: The average waiting times of the four reconstruction strategies. The number of data objects stored in one stripe varies from 1 to 2,048.

An intriguing observation drawn from Fig. 3.17 is that when the stripe capacity is low (e.g., 1), the *clustering* system offers no performance improvement over the greedy system. Nevertheless, the performance improvement of the *clustering* system over the *greedy* one become more pronounced when the stripe capacity keeps rising. Is is worth mentioning that with the increment of stripe capacity, the average waiting time of the greedy system is worsened. For example, in Fig. 3.17(a), when the stripe capacity is configured to 16, the average waiting times of *greedy*, *clustering*, *recommendation*, and *POST* are 4170, 3494, 3049, 2611, respectively. In case of stripe capacity being 2048 in Fig. 3.17(c), the average

waiting times of the four schemes are 272, 197, 192, 161, respectively. When the stripe capacity is expanding, it spends more time in recovering each stripe; therefore, the average waiting time becomes higher in the Greedy and Recommendation scenarios.

The experimental results plotted in Fig. 3.17 unravel that the *recommendation* system offers a stable reduction in terms of average waiting time thanks to the system's constant predict accuracy. On the other end of the spectrum, the *clustering* system is conducive to effectively reducing the average waiting time. Such waiting-time reductions are contributed by a large number of similar data objects archived in a single stripe. In the case of POST, popular data are more likely to be discovered and set on the top of the reconstruction list.

We observe from Figs. 3.17(a) and 3.17(b) that the clustering algorithm makes little impact on user waiting time if the stripe capacity is fairly small. On the other end of the spectrum, a large stripe capacity optimizes the clustering performance in the context of data reconstruction. For instance, Figs. 3.17(b) and 3.17(d) reveal that the clustering algorithm lowers user waiting time during the data reconstruction process. In contrast, the recommendation algorithm is unable to cut back user waiting time even we enlarge the stripe capacity.

In summary, the stripe capacity make very little impacts on the *recommendation* system's performance; the POST and *clustering* systems enjoy performance improvements by scaling up the stripe capacity.

3.5 Summary

In this chapter, we developed an erasure-coded storage system called POST, which seamlessly integrates the efficient data archival and online reconstruction techniques. We implemented a k-prototype clustering controller to archive unpopular data that attract a limited number of accesses. Our POST system is reliant on the clustering controller to group files into multiple clusters, in each of which files share similar features.

In POST, we incorporated user-based collaborative filtering to deal with online data reconstruction in which faulty data nodes are rebuilt while responding to I/O requests. POST is conducive to recovering faulty nodes while boosting read performance for requests accessing data residing on the faulty nodes. We implemented a prediction module where a list of popular data is projected by keeping track of historical I/O accesses. This popular-data list provides predictions on files that are likely to be accessed in the not-too-distant future. The prediction module in POST is adept at computing similarities among users, thereby promoting popular data to be reconstructed prior to unpopular ones.

We implemented our POST system in an erasure-coded storage cluster driven by real-world datasets. We conducted extensive experiments to demonstrate that the POST system is adroit at reducing user waiting time by the virtue of an optimized popularity-aware reconstruction list.

In particular, we showed that weights (a.k.a., predicted ratings) are proportional to the prediction accuracy, because the correlation between the weight value and prediction accuracy offers a guideline to system administrators to optimize recommendation lists that lead to a reconstruction list. The evidence confirms that POST is capable of boosting the prediction accuracy, which significantly shortens the average waiting time.

The empirical results unveils that an increasing number of users in POST leads to a low average waiting time. Moreover, increasing the number of requests per user allows POST to deliver outstanding performance improvements over the three alternative schemes (i.e., the *greedy*, *clustering*, and *recommendation* systems), because popular data in POST are more likely to be discovered and set on the top of the reconstruction list.

Chapter 4

The Popularity-Aware Cache Replacement

In this chapter, We propose a popularity-driven caching policy referred to as *PDC* in this study, which leverages future accesses predictions to push cache-replacement performance to the next level for big data applications. Our PDC advocates for data recommendation algorithms to gauge popularity values for data objects from active users' access history. Popularity values in turn signify data replacement priorities amid making replacement decisions. In addition to orchestrating active users (e.g., log into the system) and inactive users (e.g., log out from the system), a management module in the PDC system keeps track of access preferences (e.g., popularity measures) of the users.

Let us organize the rest of this chapter as follows. The design details of PDC system are described in Section 4.1. Section 4.2 presents the user-based collaborative filtering recommendation algorithm. After elaborating performance metrics and experimental settings, Section 4.3 offers the performance analysis in terms of hit ratio and byte hit ratio. We summarize our achievements in this part of the dissertation in Section 4.4.

4.1 System Design

In this chapter, we elaborate on the overall system design of the proposed *PDC* system powered by the mathematical underpinnings. We also shed some light on the functionality of the core modules that are in charge of making recommendations and caching popular data.

4.1.1 Overview

Fig. 4.1 depicts the high-level system architecture of *PDC*. Our proposed PDC system orchestrates caches, a catalog of content items (e.g., movies) and a set of historical user

access information. Throughout this chapter, we refer to the content set and the user-access-information set as O and U, respectively.

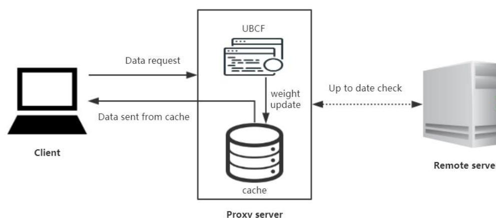


Figure 4.1: The high-level architecture of the PDC system.

The caches in PDC are limited resources storing popular data to be accessed in the not-too-distant future. In the PDC system, the caches are deployed on a back-end server. Thanks to large-scale cluster computing systems, we construct a large cache region using distributed main memory in a storage cluster. The large cache resources are expected to store popular data along with the entire catalog (a.k.a., metadata).

Big content items are stored in a storage cluster, where parallel disk systems furnish high I/O throughput. In addition to content items, metadata in the form of tags are managed by PDC on the storage cluster. A wide range of thematic categories (e.g., movies, music, news) are stored and archived in the storage cluster.

After users accessing the contents managed by PDC, the corresponding accesses are treated as historical data captured in the user-access-information set U. To speed up slow I/O processes, PDC attempts to keep popular data in the large caches to avoid unnecessary disk I/Os. As such, users are enabled to quickly access data contents residing in main-memory caches rather than disks. Upon the arrival of a user request, PDC determines if the requested data items are kept in the memory caches.

When a user logs into a storage system, the recommendation module (see Section 4.1.2) originates a recommendation list guiding the user's future accesses. Each recommendation list is comprised of a group of data items coupled with the user's interest levels in the data. Thanks to the fact that recommendation lists can be made available prior to users' access, the storage system avoids extra overhead and resources dedicated to calculating the

lists on the fly. PDC aims at curtailing the I/O accesses of active users and; therefore, recommendation lists of active users should be an input of PDC. In other words, PDC makes cache replacement decisions using the recommendation lists of active users rather than inactive ones. To implement this strategy, we allow PDC to backup the recommendation list of a user when the user logs out of the storage system; the recommendation list is restored if the user logs in the future.

The PDC gauges all active users' interest levels with the perspective of data items in the storage system. A data item's popularity is derived from all users' interest levels with respect to the data. PDC manages a *popularity list*, which contains the popularity measures of all data items to be accessed by active users. The length of the popularity list largely depends on the cache size and the capacity of the entire storage system. A large storage system with a big cache size tends to have a long popularity list and vice versa.

The cache replacement policy governed in PDC is driven by the popularity list. The overarching goal of PDC is to evict the least popular data from the cache while reserving cache resources for popular ones. The popularity list is dynamically changing and the reason is two fold. First, user states are transitioning between active and inactive. Second, user interests are continuously shifting. Hence, the popularity values of data are fluctuating. From the perspective of active users, the former popular data may become warm or cold. Data items are unpopular because the number of active users accessing the data is reduced below a certain threshold.

For simplicity without loss of generality, we assume the preferences and interests of individual users are relatively stable. We make use of a Poisson process - an independent reference model - to capture the arrivals of I/O access issues by active users. It is worth noting that the independent reference model, simulating access behaviors, has been widely adopted in the research community (see, for example, [16], [127]).

4.1.2 The Popularity Calculator Module

The popularity-aware cache replacement strategy imports the idea of popularity calculator from POST. The UBCF algorithm is adroit at predicting future user accesses derived from people's similar preferences and interests. We argue that user preferences are predictable and the reason is two-fold. First, the user preferences and interests are stable within a certain time period. Second, future preferences are strongly correlated to past preferences. Similar justifications can be found in an early study [133].

The UBCF algorithm predicts a user's future preferences or interests by (1) comparing the user's access behaviors with the historical accesses of the other users and (2) searching the nearest neighbors to project the future preferences. We select the top h data objects that have the highest predicted level of interest to construct a recommendation list. Then, all the recommendation lists are aggregated into a single popularity list based on the data objects' integral levels of interests. If users request data object is not in the cache, the least unpopular data will be replaced first.

4.1.3 Cache Replacement

Caching system plays an essential role in improving the performance of web-based systems (e.g., minimizing the utilization of network bandwidth, decreasing user-perceived delays, and reducing workloads). Three factors that have a high impact on caching system include cache consistency, cache pre-fetching, and cache replacement [6] [63] [60]. Among the three elements, the cache replacement method is placed under the spotlight in this dissertation study. We demonstrate a way of leveraging a cache replacement scheme to significantly enhance the performance of storage systems powered by our PDC (see also Section 4.2).

When a cache is fully loaded with data objects, a replacement strategy is kicked in to manipulate the cache to release space for newly arrived objects. The primary objective of an ideal cache replacement policy is to evict undesired objects that are unlikely to be accessed

in the not-too-distant future, thereby optimizing the utilization of the cache with a high cache hit rate.

In large-scale storage systems, data objects sharing similar features are stored in identical-sized blocks. As such, we are devoted to maximizing cache replacement efficiency for homogeneous data objects that are considered to have similar sizes, types, and structures. In this case, the features of data objects generally are not a key affecting cache replacement decision.

4.2 A Generalized Popularity-Driven Cache Replacement Algorithm

This section describes the conceptual and mathematical underpinnings of the cache replacement management model developed for establishing the PDC system. We start in this section by introducing the UBCF algorithm. Next, we formally define the popularity list extracted from UBCF and present the cache replacement algorithm for the big-data storage system. To facilitate the presentations of the algorithms, we summarize the symbols and notation used throughout this manuscript in Table I.

4.2.1 User-Based Collaborative Filtering

The Fig.4.2 shows the workflow of PDC module. During the course of cache replacement, data accessed by users are fetched from nodes to the cache in the system. The collaborative filtering algorithm deployed in our PDC system makes recommendations on future accessed data objects (a.k.a., data contents) to active users by comparing ratings in terms of the levels of interests harvested from similar users. Statistical evidence (see, for example, [101][47]) unveils that recommended data objects are more likely to be accessed by users than those non-recommended ones and; therefore, PDC leverages the popularity list coupled with user-request lists as a guideline to manage data's popularity. In doing so, data with a strong access likelihood are kept in the cache without being evicted by the cache replacement.

Table I: Symbols and Notation

Symbol	Annotation
O	a dataset includes a set of data objects
o_i	the ith data object in dataset O
e_i	key of the key-value pair in ith data object
v_i	value of the key-value pair in ith data object
r_{xi} r_{yi}	rate value of user x or y on data object i
U	a set of users are loging in or loging out the system
$sim(u_x, u_y)$	the similarity value between users x and y
R_{xy}	a set of data objects rated by both user x and user y.
r_{xi} r_{yi}	rate value of user x or y on data object i
S	records the set which collects the q' nearest neighbors similarity set
u_x u_y	the x/yth user in the set U
W	a set of predicted ratings value
$w_{x,i}$	the predicted rating of data object o_i for user u_x
$\phi_{x,i}$	an object-popularity pair records the data object o_i and the corresponding popularity $w_{x,i}$
$p(u_x, o_i)$	the predicted ratings $w_{x,i}$ of data object o_i for user u_x
$P(u_x, h)$	records h highest recommended data objects o_i and corresponding popularity in $p(u_x, o_i)$ for u_x
L	records the single popularity list merged by multiple recommendation lists

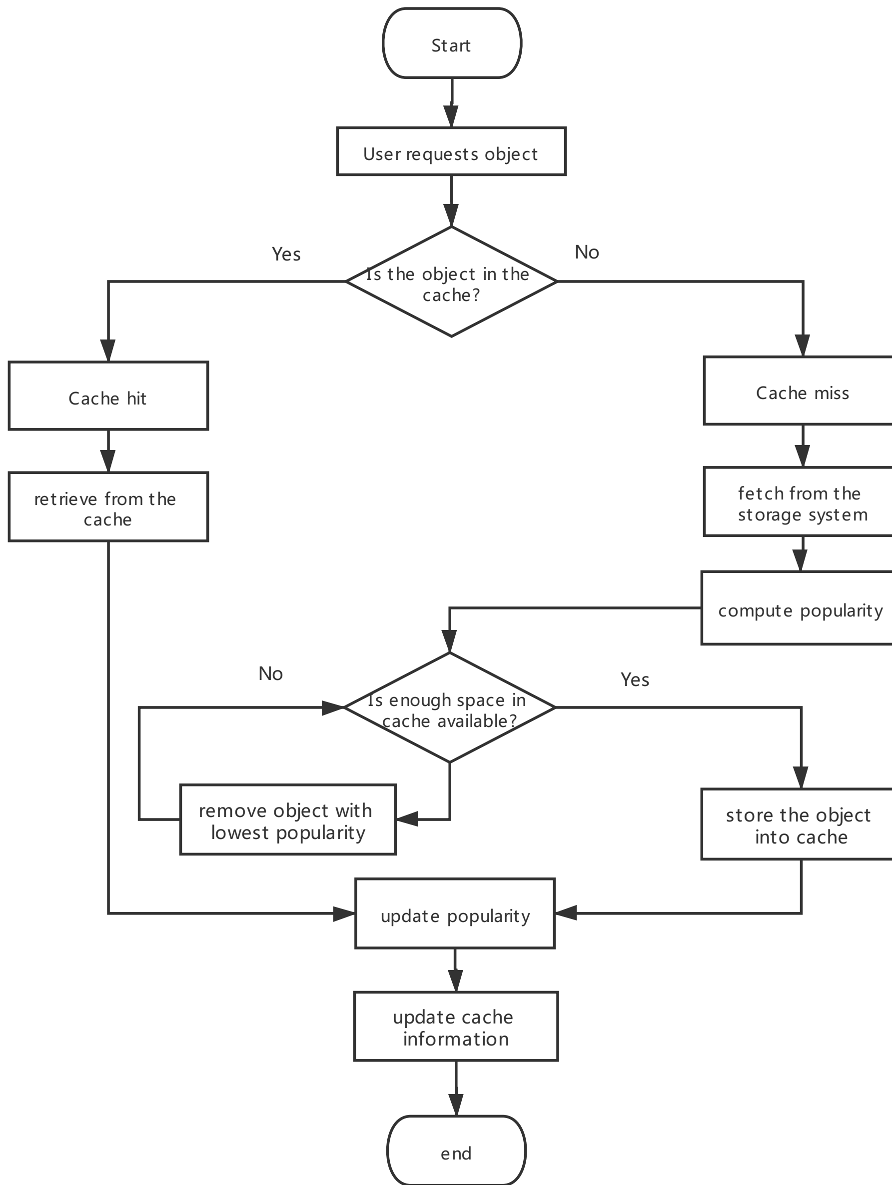


Figure 4.2: The work flow of PDC cache replacement.

We extend and incorporate the *user-based collaborative filtering* algorithm [133][47], which delivers striking performance in creating recommendation lists by computing similarity values between any pair of two users in terms of content ratings (a.k.a., user-item ratings). The algorithm embraces two steps below in originating a recommendation list:

- Step 1: To construct a user set in which users share similar interests to an active user.
- Step 2: To project the active user's interested data that this active user has never accessed in the past.

Our PDC system originates and maintains user-item ratings as part of user profiles, which are organized in the format of a user-item rating table (see also the input component in Fig. 3.5). A rating table consists of a matrix, where each row and each column represents a user and a data object, respectively. The value sitting at the intersection of a row and a column denotes a user rating. For example, in the rating table plotted in Fig. 3.5, the intersection between user u_x and data object o_i indicates the rating of user u_x on object o_i.

The collaborative filtering algorithm gauges the similarities among users with respect to user-item ratings, thereby making recommendations to active users in accordance with the observed behaviors of similar users. Similarities can be measured in various approaches like *Cosine*, *Pearson*, and *Euclidean* [133]. In our PDC system, we advocate for the *Pearson correlation coefficient* algorithm to quantify similarities between any pair of two users u_x and u_y in the user set U. More formally, the similarity between u_x and u_y is expressed as

$$sim(u_x, u_y) = \frac{\sum_{i \in R_{xy}} (r_{xi} - \bar{r}_x)(r_{yi} - \bar{r}_y)}{\sqrt{\sum_{i \in R_{xy}} (r_{xi} - \bar{r}_x)^2} \sqrt{\sum_{i \in R_{xy}} (r_{yi} - \bar{r}_y)^2}}. \tag{4.1}$$

where $\bar{r}_x$ and $\bar{r}_y$ are the average ratings from users u_x and u_y, r_{xi} and r_{yi} are ratings on data object o_i recorded by users x and y. R_{xy} is a set of objects rated by users x and y. z

The algorithm aggregates the q' nearest neighbours based on the similarity measures quantified by Eq. 4.1. Hence, the similarity set is written as

$$S = \{sim(u_x, u_1), sim(u_x, u_2), ..., sim(u_x, u_{q'})\}, q' < q. \tag{4.2}$$

where q' is the number of nearest neighbours of user x, q is the number of users in the system and $sim(u_x, u_i)$ is the similarity between target user u_x and user u_x's neighbour u_y.

A predicted rating $p(u_x, o_i)$ is calculated as a weighted average (see also the second item on the right-hand side of Eq. 4.3) of neighbour's mean ratings plus active user u_x's mean rating $\bar{r}_x$. An array of users tend to offer high or low ratings to all evaluated data objects and; therefore, user biases do exist. The predicted rating of user u_x on object o_i is written as

$$p(u_x, o_i) = \bar{r}_x + \frac{\sum_{y \in H} (r_{y,i} - \bar{r}_y) sim(u_x, u_y)}{\sum_{y \in H} sim(u_x, u_y)}, \tag{4.3}$$

where $sim(u_x, u_y)$ is the similarity between users u_x and u_y, $\bar{r}_x$ and $\bar{r}_y$ are embedded to suppress user-associated biases. We refer to the predicted rating $p(u_x, o_i)$ as popularity (a.k.a., level of interest) $w_{x,i}$, which is expressed below.

$$w_{x,i} = p(u_x, o_i). \tag{4.4}$$

A data object exhibiting large popularity is a popular data object. Intuitively, an enormous number of requests accessing highly popular objects in storage systems. We make use of object-popularity pairs as a key data structure for recommendation lists. Let us consider object o_i and its popularity $w_{x,i}$ with respect to user u_x. We define object-popularity pair $\phi_{x,i}$ of u_x and o_i below.

$$\phi_{x,i} =< o_i, w_{x,i} > . \tag{4.5}$$

We denote $P(u_x)$ as a recommendation list of u_x. With Eq. 4.5 in place, we derive the predicted ratings of user u_x on all the data objects $\{o_1, o_2, ..., o_m\}$. Now, we express the set of predicted ratings of u_x as $\{\phi_{x,1}, \phi_{x,2}, ..., \phi_{x,m}\}$, which are stored in set $P(u_x)$ in an non-increasing order. Thus, u_x's sorted predicted-rating set $P(u_x)$ is modeled as

$$P(u_x) = \{\phi_{x,1'}, \phi_{x,2'}, ..., \phi_{x,m'}\}, \quad \text{where} \quad w_{x,1'} \geq w_{x,2'} \geq ... \geq w_{x,m'}. \tag{4.6}$$

Recommendation list $P(u_x, h)$ of user u_x contains the top h ratings in the sorted predicted-rating set $P(u_x)$. Hence, recommendation list $P(u_x, h)$ is written as

$$P(u_x, h) = \{\phi_{x,1'}, \phi_{x,2'}, ..., \phi_{x,h'})\}. \tag{4.7}$$

We implement the filtering algorithm using cutting-edge in-memory computing to speed up the process. Please refer to the literature (e.g., [118]) for similar in-memory computing schemes tailored for big-data applications.

4.2.2 A Popularity-Based Cache Replacement Policy

In our proposed system, a recommendation list (see also Eq. 3.14) is customized for each active user accessing the system. The PDC cache replacement controller merges the recommendation lists of multiple users into an aggregated popularity list. Let l_i be the popularity measure of the ith data object. Popularity l_i is derived from popularity $\phi_{x,i}$ of object i from the perspective of user x. We quantify popularity l_i as a summation of all the popularity measures from the view points of all the users accessing object i. Thus, popularity l_i is written as

$$l_i = \sum_{x \in U} (\phi_{x,i}), \quad \phi_{x,i} \in P(u_x, h). \tag{4.8}$$

Formally, the popularity list can be expressed as

$$L = \{l_1, l_2, ...l_i, ..., l_n\} l_i \in O, \tag{4.9}$$

where l_i contains the meta data object o_i and the corresponding popularity value w_i.

Algorithm 3 depicts the procedure of PDC's cache replacement policy, which is fed with an aggregated popularity list L merged from individual recommendation lists. The cache replacement policy is kicked in when objects are evicted from the cache to release space to accommodate new items. The input information of Algorithm 3 embraces historical user I/O accesses coupled with ratings, which are captured in popularity list L (see also Eq. 4.9).

Algorithm 3: The popularity-based cache replacement policy.

Input:
  Popularity list L; /* see also Eq. (4.9) */
  All the data objects o_i in cache C;
  User's request $r(o_{i'})$;
Output:
  An updated cache C.

1: **if** $o_{i'} \notin C$ **then**
2:   **for** $o_i \in C$ **do**
3:     $l_i = (o_i).getValue()$
4:     **if** $min < l_i$ **then**
5:       $min = l_i$;
6:       $least_pop = i$;
7:     **end if**
8:   **end for**
9: **end if**
10: cache.pop(o_{least_pop})
11: cache.add(o_{new})
12: **return**

Algorithm 3 starts by checking if new request $r(o_{i'})$ attempts to access object $o_{i'}$ that is residing in cache C (see *Step 1*). The algorithm continues its execution only if the requested object $o_{i'}$ has not yet been buffered in the cache. *Steps 2-6* traverse the current data objects stored in cache C in order to pinpoint the least popular one to be evicted. More specifically, *Step 6* keeps track of the object (i.e., o_{least_pop}) with the lowest popularity in the cache.

Function $getValue()$ in *Step 4* extracts popularity value l_i (see also Eq. (4.9)) of object o_i . The last two steps (i.e., 10 and 11) substitute the new object o_{new} for the least popular one o_{least_pop} in cache C. In doing so, the cache is in the right position to maintain the most popular objects for future user requests.

Now we are positioned to elaborate the design of the popularity-list updating algorithm (see also Algorithm 4), which is instantly invoked each time when there is a new user log into the system. This algorithm meticulously updates the popularity list by merging multiple new recommendation lists (e.g., recommendation list $P(u_x, h)$ for user u_x), which is derived from Eq. 3.10. On the other end of the spectrum, the popularity-list updating algorithm is triggered if a user logs out of the system. Algorithm 4 formally describes the procedure of updating the popularity list upon the arrivals and departures of users. The input parameters of Algorithm 4 include an existing popularity list L and the recommendation list $P(u_x, h)$ of newly arrived user u_x. The output parameter is an updated popularity list referred to as L'.

Algorithm 4: The controller of popularity update.

Input:

Popularity list L; /* see Eq.4.9*/

The recommendation list $P(u_x, h)$; /* the user may coming or leaving */

Output:

`// for all object-popularity pairs in set` $P(u_x, h)$

1: **for** $\phi(x, i') \in P(u_x, h)$ **do**
2: $o_{i'} = get_object(\phi(x, i'))$
3: **for** $l_i \in L$ **do**
4: $o_i = get_object(l_i)$;
5: **if** $o_{i'} == o_i$ **then**
6: $w_{x,i'} = get_popularity(\phi(x, i'))$;
7: $w_i = get_popularity(l_i)$;
8: $w_i = w_i + w_{x,i'}$;
9: $L' = put_popularity(L, w_i)$;
10: **end if**
11: **end for**
12: **end for**
13: **return** L'

Step 1 in Algorithm 4 traverses each object-popularity pair $\phi_{x,i'}$ in set $P(u_x, h)$, whereas *Step 3* processes all the object-popularity pairs (e.g., l_i) in popularity L in a for loop. *Steps 2* and *4* acquire objects $o_{i'}$ and o_i from recommendation list $P(u_x, h)$ and popularity list L by invoking function *get_object*. It is worth noting that function *get_object* in *Step 2* returns an object from an input object-popularity pair (e.g., $\phi(x, i')$), whereas *get_object*() in *Step 4* renders an object from a given input popularity list (e.g., L). Step 5 checks the consistency between data object $o_{i'}$ and o_i obtained from *Steps 2* and *4*. More specifically, if $o_{i'}$ and o_i are identical, then popularity list L' should be updated by *Steps 6-9* as follows. *Step 6* retrieves popularity value $w_{x,i'}$ from pair $\phi(x, i')$. Similarly, *Step 7* obtains overall popularity value w_i from pair l_i in set L. *Step 8* aggregates popularity value $w_{x,i'}$ to the corresponding overall popularity w_i for the data object o_i. When user x logs into the system, popularity $w_{x,i'}$ is a positive value. $w_{x,i'}$ turns out to be a negative one, if user x logs out of the system. *Step 9* determines an updated popularity list L' from the former list L and the updated popularity value w_i computed in *Step 8*. *Step 13* returns updated popularity list L to a caller.

4.3 Performance Evaluation

As part of an empirical study, we quantitatively evaluate the performance of the proposed PDC system. We emulate a storage system that maintains a collection of O of data items, which are concurrently accessed by a number of clients.

4.3.1 Performance Metrics and Experimental settings

Predominant performance metrics adopted to quantify performance of cache replacement algorithms are hit ratio (i.e., *HR*) and byte hit ratio (i.e., *BHR*). Hit ratio measures the percentage of requested data that are residing in the cache; byte hit ratio stipulates a ratio of bytes served by the cache over the total number of bytes requested by users. HR and BHR measures might have a stark difference in scenarios where only a few but large data objects are stored in the cache. We quantitatively evaluate the performance of PDC using a

real-world dataset containing movie data objects [20]. The movie dataset records user ratings accompanied by users' access history logs. More specifically, the dataset is comprised of 26 million ratings from a total of 270,000 users for all the 45,000 movie items.

We place this real-world dataset under the spotlight, because movie data systems (e.g., Netflix) manage user access histories coupled with preferences. To carry out the extensive experiments, we randomly select 20 movies from each user's access history to construct access workload. We configure two system parameters - cache size and cache replacement policy - to test the impacts of the parameters on system performance.

User access patterns are collected and resembled in the movie dataset. We evaluate the effectiveness of PDC that utilizes the access patterns of movies from a large number of online users. Cache replacement decisions are dynamically and judiciously made by PDC based on user access patterns.

We configure cache size at a total of 12 levels, ranging in a window of between 2 GB and 2048 GB. The upper bound of the cache size is 2048 GB, which is sufficiently large to accommodate an entire document set for any tested I/O traces. Therefore, the cache size of 2048 GB resembles an extreme scenario where the cache size is unbounded.

In recent years, a handful of proxy cache replacement policies have been proposed to enhance the I/O performance of storage systems (see, for example, [7], [122], and [8]). To make our extensive experiments manageable, we pay particular attention to the three replacement policies, including the LRU, LFU, and PDC algorithms. The rationale behind selecting these caching algorithms is that the competitors are representatives of a broad range of cache replacement policies, namely, recency-based, frequency-based, and popularity-aware strategies, respectively.

4.3.2 Overhead Analysis

Files are organized and stored in form of large blocks to cut back meta-data management overhead. It is worth noting that file systems (e.g., Hadoop distributed file system and Google

file system) built for big-data applications advocate for large blocks. Metadata in PDC is tailored to monitor the popularity measures of data blocks. The meta-data manager keeps track of the popularity levels and tags of data objects stored in the PDC system. It takes four bytes to hold the popularity and tag of each data object. Therefore, the meta-data management overhead in PDC largely depends on the number of data objects and data object size in a storage system. More broadly, the meta-data space overhead of each data object is as small as four bytes and; thus, the overhead ratio is quantified as a ratio between four bytes and the data object size. Hence, we have the overhead ratio of $\frac{1}{object-size}$. For example, let us suppose the data object size is set to 64 MB and the number of data objects is 1000; the space overhead of the metadata is 4 KB. In this case, the overhead ratio is 1/16M, which is negligible.

With an increasing number of users, the probability of multiple users accessing the same data goes up accordingly. As such, the average waiting time tends to be shortened thanks to multiple requests accessing shared popular data. PDC induces fairly low computation overhead because recommendation lists are pre-processed prior to the caching procedure. The sorting algorithm - a major contributor to PDC's overhead – has its time complexity capped at O(n log (n)). A handful of cache replacement strategies enjoy the time complexity lower than that of PDC. Nevertheless, PDC's overhead caused by the sorting algorithm is mitigated through two venues. First, such an overhead is remarkably offset by the performance gain of a high cache hit ratio, which considerably slashes I/O access time. Second, recommendation lists orchestrated in PDC can be originated in an offline fashion, meaning that the recommendation model is in full swing before caching occurs.

It is arguably true that PDC's scheduling time overhead in handling cache replacement is fairly low, because the system calculation for the popularity value of data object is managed by metadata. The time overhead is the time spent in computing popularity value, which is processed in the PDC algorithm 4. To be specific, the time complexity of creating popularity is $O(n^2)$, where n is the number of data objects in the recommendation list. It is worth noting

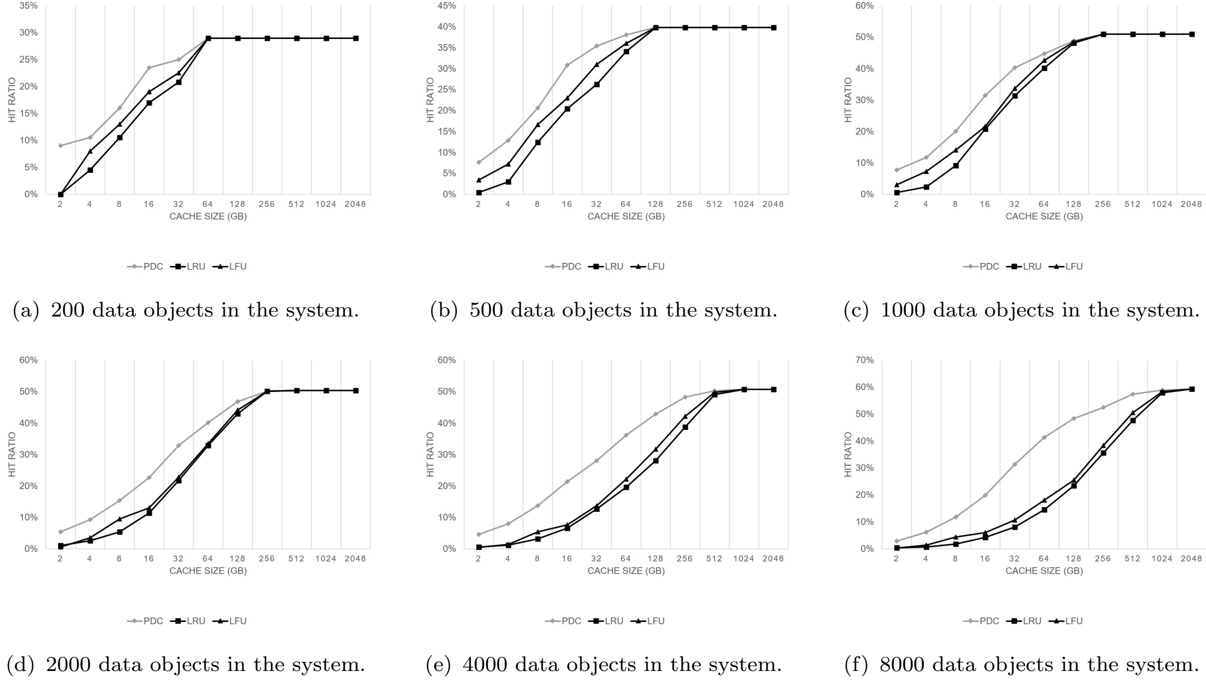
that the value of n configured in the PDC system is a relatively small value (e.g., 10). Thus, the overhead of computing data popularity is acceptably low.

4.3.3 Hit Ratio Analysis

In this group of experiments, we measure the hit-ratio performance of the popularity driven caching system. Recall that the PDC system incorporates the *UBCF* algorithm to sense user-level of interests in data objects (see also Section 3.1.4). We quantitatively evaluate the effects of the number of access requests from user groups on the performance of the three cache replacement strategies. All the user requests are randomly selected from the user access logs (a.k.a., history records). In particular, we select the number of data objects anywhere between 200 and 8000 from the log files to place on an access list; the access sequence is orchestrated by the timestamps in the source dataset. Fig. 4.1 unravels the hit ratio and byte hit ratio of our PDC and the two counterparts. The cache size in the experiments ranges from 2 GB to 2048 GB.

Comparing all the six sub-figures in Fig. 4.3, we observe that with the increasing number of accessed data objects, the maximum hit rate surges from 30% (see Fig. 4.3(a)) to 60% (see Fig. 4.3(f)). Expanding the number of data objects benefits hit ratio performance thanks to the following two factors. First, statistical trends show that 80% data only capture less than 20% of accesses. Second, when the number of users or requests increases, the popular data objects enjoy a high possibility to be repeatedly accessed in the cache. Theoretically speaking, it is impossible to push the maximum rate all the way to 100%, because an array of movies must be fetched to the cache due to either cold start or cache replacements.

The results plotted in Fig. 4.3 reveal that PDC and LFU are superior to LRU, because PDC and LFU preserve frequently requested data objects in the cache while evicting unpopular ones. LRU achieves the lowest data-object hit ratios among the three evaluated policies. When the cache size is set to a fairly low value (e.g., 2 GB), the hit ratios of LRU and LFU are close to zero. In the case of small cache sizes, the performance of PDC is approximately



(a) 200 data objects in the system. (b) 500 data objects in the system. (c) 1000 data objects in the system.

(d) 2000 data objects in the system. (e) 4000 data objects in the system. (f) 8000 data objects in the system.

Figure 4.3: The impacts of cache size on the hit ratio performance of the three cache replacement policies. The cache size ranges from 2 GB to 2048 GB; the number of data objects is set to 200, 500, 1000, 2000, 4000, and 8000.

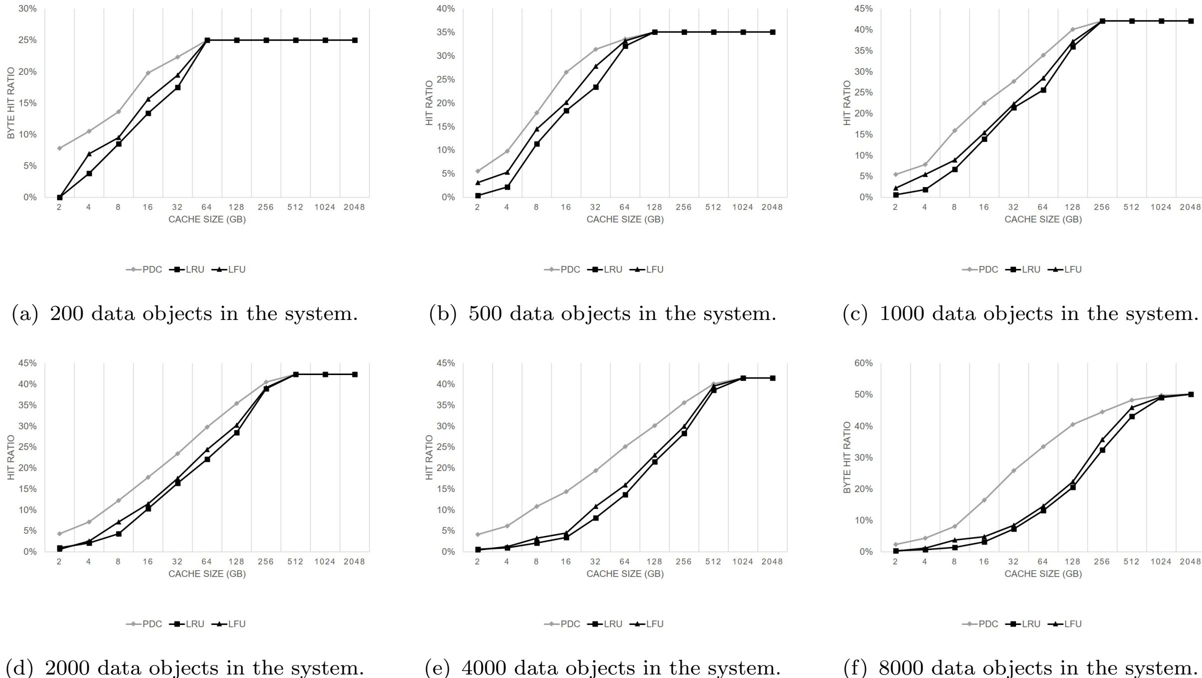
10 times better than those of LRU and LFU. PDC's outstanding performance is expected, because PDC makes it more likely to share data objects among the active users under a small cache size. When the size of cache rises, PDC is still a front runner even though the hit-ratio gap between PDC and the two counterparts is narrowed. For example, when the size of the cache is set to 4 GB (see Fig. 4.3(b)), the hit ratio in PDC, LRU and LFU are 12.8%, 3%, and 7.2%, respectively. In the case of 32 GB, the hit ratios in the three cache replacement policies climb to 38%, 34.1%, and 36%, respectively. In contrast, Fig. 4.3(e) illustrates that the hit ratios of the three policies are 11.7%, 1.8%, 4.3% under the cache size of 4 GB and 52.4%, 35.6%, 38.3% under the cache size of 128 GB. This trend suggests that expanding cache size diminish PDC's advantage over the conventional LRU and LFU.

Comparing the performance results depicted in Figs. 4.3(e) and 4.3(a), we observe that scaling up the system in terms of the number of data objects boosts the hit ratio performance of all three policies. The rationale behind such an improvement is that popular data objects

have an increased possibility to be accessed when the dataset is expanded. PDC collects the popularity information from the user access logs, aiming to prune unpopular data from the cache. With the increment of a number of users, the chance of multiple users accessing the same popular data goes up accordingly. As such, the hit ratios spike thanks to the popular data shared among the requests.

4.3.4 Byte Hit Ratio Analysis

This group of experiments is focused on assessing byte hit ratios of the *PDC*, *LRU*, and *LFU* cache replacement policies under various cache sizes. Fig. 4.4 illustrates the impacts of cache size on the byte hit ratios of the three policies when the number of requested data objects ranges from 200 to 8000.



(a) 200 data objects in the system.

(b) 500 data objects in the system.

(c) 1000 data objects in the system.

(d) 2000 data objects in the system.

(e) 4000 data objects in the system.

(f) 8000 data objects in the system.

Figure 4.4: The impacts of cache size on the byte hit ratio performance of the three cache replacement policies. The cache size ranges from 2 GB to 2048 GB; the number of data objects is set to 200, 500, 1000, 2000, 4000, and 8000.

Fig. 4.4 reveals that *PDC* achieves the highest byte hit ratio among the three competitors, whereas *LRU*'s performance is in the last place. *PDC* is a winner across the board

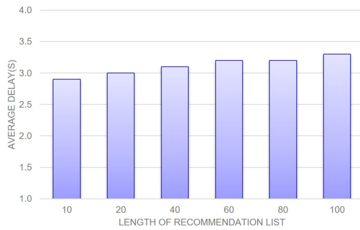
(e.g., small and large objects) because *PDC* judiciously predicts future accesses using the popularity values rather than historical patterns. Evidence confirms that popularity levels are a superior tool than access logs to project future I/O access patterns. Moreover, *PDC* retains popular data objects in the cache for a long time period without discriminating against large data objects. When the amount of shared data is expanded, the byte hit ratio tends to noticeably benefit from popular data objects. In other words, with the increment of number of users and data objects, the probability of multiple users accessing the same data goes up accordingly.

In a nutshell, *PDC* acquires and manages popular data information to enhance the cache replacement policy performance. For the aforementioned reasons, PDC delivers better hit ratios as well as byte hit ratios than the other two counterparts.

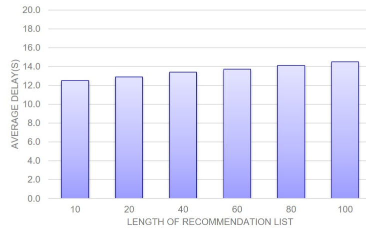
More importantly, the experimental results unravel that increasing the number of accessed data objects (see Fig. 4.4(a)-(f)) enables*PDC* to significantly boost performance with respect to byte hit ratio. This trend is reasonable, because a big set of data objects accessed by users benefit the accuracy of popularity predictions. Regardless of the number of data objects, *LRU* exhibits the worst performance among the three replacement policies. The performance gap between *PDC* and *LRU* is widened when there is not a weak correlation between past and future access patterns under a large number of requests in an I/O cache system.

4.3.5 Impacts of Recommendation List Length

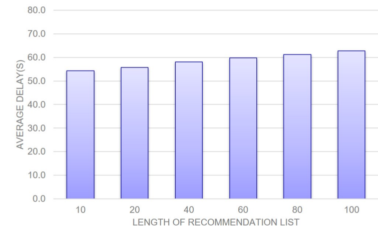
Now we are positioned to assess the sensitivity of time delay and hit ratio on the length of recommendation list (see also h in Eq. 3.14). To achieve this goal, we vary the recommendation-list length from 10 to 100 while keeping the cache size at 32 GB. Fig. 4.5 illustrates the impacts of the recommendation list length on the average time delay and hit ratio in PDC.



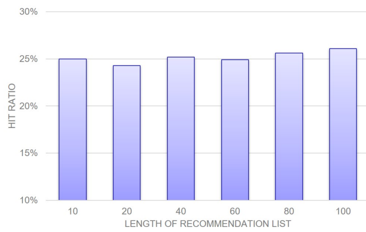
(a) average delay while 200 data objects in the system.



(b) average delay while 1000 data objects in the system.



(c) average delay while 4000 data objects in the system.



(d) hit ratio assessment while 200 data objects in the system.

(e) hit ratio assessment while 1000 data objects in the system.

(f) hit ratio assessment while 4000 data objects in the system.

Figure 4.5: The impacts of recommendation-list length on the average time delay and hit ratio performance of the PDC cache replacement policy. The cache size is set to 32GB; the recommendation list is set to 10, 20, 40, 60, 80, and 100; the number of data objects is set to 200, 1000 and 4000.

An intriguing observation drawn from Fig. 4.5 is increasing recommendation-list length h imposes mixed impacts on time delay and hit ratio. Regardless of the performance gain or degradation, length h makes marginal effects rather than dramatic ones. In Fig. 4.5(e), for instance, when the length is set to 20, 60, and 100, the average time delay of PDC is 12.9, 13.7, and 14.5 seconds, respectively. This trend is reasonable because PDC pays a small cost (1) to select the top h objects and (2) to execute the cache replacement algorithm. The following factors inspire us to maintain a short recommendation list in PDC.

- Due to the limited cache size, only top candidates in the recommendation list length are stored in the cache. An ideal recommendation list should be sufficiently long to contain objects that are likely to be kept in the cache.

- Data objects at the bottom of a recommendation list tend to bear low-weight values, which have little influence on the performance.

- An extremely large list embracing a large number of unpopular objects makes no contribution in enhancing cache hit ratio.

- A long recommendation list inevitably escalates I/O cost. Managing the list at the most appropriate length can remarkably suppress cost.

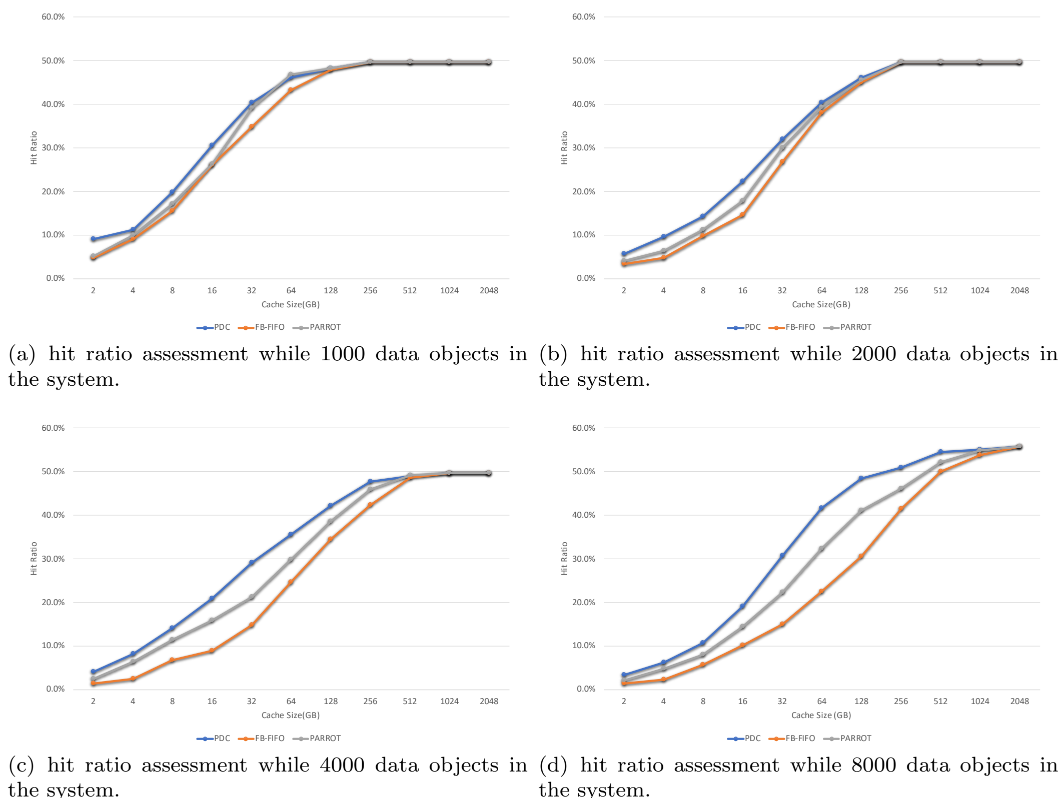
Thanks to the fact that the recommendation lists are made available prior to user accesses, the time delay is observed before kicking in cache replacement. Hence, a slight increase in time delay has few adverse impact on PDC.

Figs. 4.5(d), 4.5(e), and 4.5(f) unravel that a long recommendation list length slightly optimize hit ratio because of improved predict accuracy. For example, in case of the recommendation length being 20, 60 and 100 in Fig. 4.5(b), the hit ratios are measured as 23.7%, 24.5% and 25%, respectively. We observe from Figs. 4.5 that although time delay and hit ratio are not sensitive to the list length h, these two performance metrics do grow marginally when length h is moving up.

4.3.6 Comparison with the Advanced Cache Replacement Strategies

In this group of experiments, we dive into the comparison between PDC and the advanced cache replacement technologies - FB-FIFO [42] and PARROT [74]. FB-FIFO relies on an analytical model to create a variable-size cache segment for data objects requested more than once within a short period. PARROT takes an imitation learning approach to automatically keeping track of cache access patterns.

The experiment settings are identical to those articulated in section 4.3.3. Because we systematically compare PDC with LRU and LFU in Section 4.3.3, this section is dedicated to the hit-ratio analysis of PDC, FB-FIFO, and PARROT. The hit-ratios of LRU and LFU are omitted from Figs. 4.3. Figure 4.6 unravels the relationships between the hit ratios of PDC, FB-FIFO, and PARROT and the cache size, which varies from 2GB to 2048 GB.



(a) hit ratio assessment while 1000 data objects in the system.

(b) hit ratio assessment while 2000 data objects in the system.

(c) hit ratio assessment while 4000 data objects in the system.

(d) hit ratio assessment while 8000 data objects in the system.

Figure 4.6: The impacts of cache size on the hit ratio performance of the PDC and two advanced technology cache replacement policies. The cache size ranges from 2 GB to 2048 GB; the number of data objects is set to 1000, 2000, 4000, and 8000

Fig. 4.6 unveils that PDC noticeably outperforms PARROT and FB-FIFO in terms of hit ratio measurement. Upon the increasing cache size, PDC benefits from an enormous number of data objects sharing similar features. Not surprisingly, the hit ratio of PDC is more sensitive to the large number of shared data objects compared with those of FB-FIFO and PARROT. Similar to PARROT, PDC exploits the machine-learning algorithm to assess data popularity. Nevertheless, PDC has an edge over PARROT in terms of computing overhead, because PDC effectively hides the overhead by invoking the machine-learning algorithm prior to sparking cache replacement. A comparison between the machine learning (PDC and PARROT) and non-machine learning strategies (FB-FIFO) demonstrates that when the number of users or accesses escalates, popular data have a high likelihood to be referenced and detected by the system. Consequently, the popular data objects tend to be kept in the cache thanks to being ranked at the bottom of the replacement list. In a nutshell, we conclude that PDC exhibits superb and scalable cache replacement performance for storage systems where it is not uncommon for a large number of users to share similar access preferences and patterns.

4.4 Summary

In this chapter, we elaborated on the design and implementation of a web-proxy-cache replacement system called *PDC*. We built a recommendation module to harvest popularity values of data objects in storage systems. Our PDC is reliant on the recommendation system to gauge data popularity, thereby evicting unpopular data from caching system.

We incorporated in PDC the *user-based collaborative filtering* technique to govern cache space in which a data-object popularity list is dynamically maintained while responding to I/O requests. PDC is beneficial to a wide range of big data applications such as video and music streaming services (e.g., Netflix). PDC is conducive to optimizing caching resources in terms of cache hit ratio. The popularity-data controller manipulates the priority levels of data objects, thereby promoting popular data to be cached prior to unpopular ones.

PDC is capable of handling multiple distributed caching proxies, which cooperate and share cache resources. Apart from managing a stand-alone web caching, PDC can be seamlessly integrated with a prefetching policy in order to dramatically improve the performance of storage systems.

We implemented a prototype of PDC in a web-proxy caching system, the performance of which is intensively tested by real-world datasets. We conducted extensive experiments to demonstrate that the PDC system is adroit at pushing up cache hit ratio by the virtue of an optimized popularity-aware list.

The empirical results unveil that PDC beats the traditional cache replacement policies - LRU and LFU - with respect to cache hit ratio and byte hit ratio. Moreover, increasing the cache size leads to improved cache hit ratios and byte hit ratios of LRU and LFU. Nevertheless, PDC still outperforms LRU and LFU in the case of large cache space, because popular data managed by PDC are more likely to be discovered and cached than those administrated by LRU and LFU.

Chapter 5

Similarity-Based DDoS Detection

We devise the DDoS detection model in this chapter that seamlessly integrates two distinct modules - a sample user generator and a similarity comparison module. The first module elects a group of high fitness users from all the legal users to build a sample user set, which is periodically updated by our system. The second module is in charge of calculating the similarities among active users and sample users. We treat a user as an abnormal user if the user's similarity measures are dramatically changed with a high percentage.

We organize the this chapter into the following sections. Section 5.1 provides an overview of DDoS attack models, detection techniques in cloud storage system and articulate various application layer DDoS detection techniques in cloud. Section 5.3 outlines our proposed similarity-based DDoS detection model. The performance evaluation of Similarity-based DDoS detection model is described in section 5.4. Section 5.5 summarize the development and findings described in this chapter.

5.1 An Overview

On demanding requirement of application DDoS detection models is to deliver accurate and quick performance amid the diagnosis of cloud storage systems. Existing anomaly DDoS detection techniques customized for cloud data storage systems are focused on monitoring online requests at the application layer. It is urgent and prudent to detect DDoS attacks among online requests because DDoS attacks make storage servers unavailable while infecting remote machines through zombie code.

An effective way of preventing attacks from application layer DDoS attack is to compare active users' current behaviors with historical behaviors recorded in a given time period.

With the assistance of such behavioral comparisons, a DDoS detector is slated to determine whether or not cloud storage servers are in risks. Another key element that improves the DDoS detection performance is overhead control. Existing anomaly-based DDoS detection techniques that apply user behavioral comparisons lack extra cost analysis [80][86]. It is evident [58] that 10% of data are accessed by 90% of users, and an DDoS attack tends to issue requests with random objects [130]. We propose a novel DDoS detection model for application layer to boost the security of online cloud storage systems by deploying a user similarity measure module.

We devise the DDoS detection model that seamlessly integrates two distinct modules - a sample user generator and a similarity comparison module. The first module elects a group of high fitness users from all the legal users to build a sample user set, which is periodically updated by our system. The second module is in charge of calculating the similarities among active users and sample users. We treat a user as an abnormal user if the user's similarity measures are dramatically changed with a high percentage.

Two reasons motivate us to advocate for the similarity-based DDoS detection strategy. First, online cloud servers are accessed and shared by a large-scale user pool. Users' historical access records offer excellent samples to discover outlier users such as bots. Second, when bots attempt to simulate human users, it is a challenge to find attack sources with a low overhead. If a DDoS detector fails in swiftly pinpointing attackers, any detection delay may pose potential security threats to the entire system and users. Our DDoS detection model ensures that active users' access behaviors pass through a thorough yet lightweight analysis.

The overarching goal of the similarity comparison module is to glean users' current and historical requests records, followed by calculating similarity measures of active users and the sample ones. More specifically, the similarity module illustrates the users' behavioral changes with respect to user requests. This module works in full capacity to dynamically track and monitor the active users' similarity metrics to sense any abnormal behaviors.

5.2 A Similarity-based DDoS Detection System

In Section 5.2.1, we start the research road map by presenting a basic idea for the development of a similarity-based DDoS detection system. Section 5.2.2 depicts a system architecture where user behaviors are gleaned to furnish similarity comparisons in cloud data storage. Then, we elaborate the key concepts and a model in Section 5.2.3. Finally, in Section5.3 we outline the high-level algorithm of the similarity-based DDoS detection system catering to optimize DDoS detection performance at the application layer.

5.2.1 The Basic Idea

In this study, we pay heed to DDoS attack detection at the application layer centered around cloud storage systems. Since any security solution customized for cloud data storage is obliged to impose a non-negligible performance overhead, we demonstrate that light-weight similarity comparison techniques are capable of enhancing the performance of a DDoS detection system deployed in modern cloud storage.

The basic idea of our system solution is to analyze users' behaviors through gauging similarity values orchestrated by a user sampling algorithm, which makes similarity decisions in a way that abnormal users are detected for isolation purpose. One of the vital factors benefiting the probability of successful intrusions is the number of access points [22]. As such, we advocate for gauging the users' abnormal behaviors by comparing with a colossal group of normal users. User behavior is classified as abnormal in either of the following two cases:

- First, there is a dramatic change in similarity values between a given user and normal users compared to the given user's history.
- Second, the number of changes in similarity values between a given user and normal users exceeds a specified threshold.

In addition to similarity comparisons, we forge a user sampling module to periodically originate a relatively small sample user set to conserve computing resources. Importantly, the group of sample users are leveraged to optimize the accuracy of anomalous and to curtail the overhead of similarity comparisons. In case the given user matches the above two metrics, our proposed similarity-based detection will mark the user as an abnormal user.

5.2.2 High-Level System Architecture

Fig. 5.1 depicts the system architecture of our proposed similarity-based DDoS detection system, where a similarity comparison manager collaborates with a user sampling module in clouds. The entire system is slated to detect abnormal users from active users' regular requests. Given online user request lists sent to a server, the detection module takes similarity measures by comparing current requests against sampled users. Similarity results will be compared with a history record to identify potential anomalies.

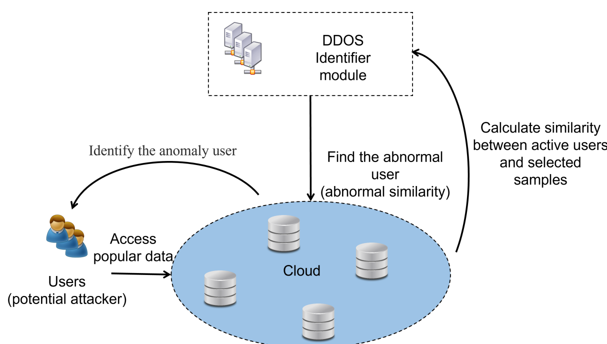


Figure 5.1: The architecture of similarity-based DDoS detection model running in clouds. abnormal users are identified through the similarity comparison.

The overarching goal of the system is to alleviate DDoS risks at the application layer by detecting anomalous requests from all the online users requests. To achieve this goal, a management module is engaged to gauge users' similarity measurements, which are maintained as critical moving parts in the metadata of users. Anomalous behaviors are likely to be taken out from users with a growing number of similarity comparisons. It is arguably true that the management module in this architecture is adroit at governing anomaly detection, where similarity comparisons are configured in accordance to tracking users' similarity

changes. With the similarity manager in place, outlier users are assigned a red flag to be handled by the security module.

5.2.3 Concepts and Key Steps

Fig 5.2 illustrates the main steps undertaken in the DDoS detection system. In a given time slot, the system checks request lists of online users. Next, the system calculates the similarities between the online users and sample users. The similarity comparison is performed in full swing after the similarity calculation is completed. We diagnose any anomalous users through the help of two thresholds: (1) the similarity value change with regard to a single sample user and (2) the amount of similarity changes with respect to multiple sample users (see Eq. 5.7). The system defines the current online user as an abnormal user when the metrics exceeds these two thresholds.

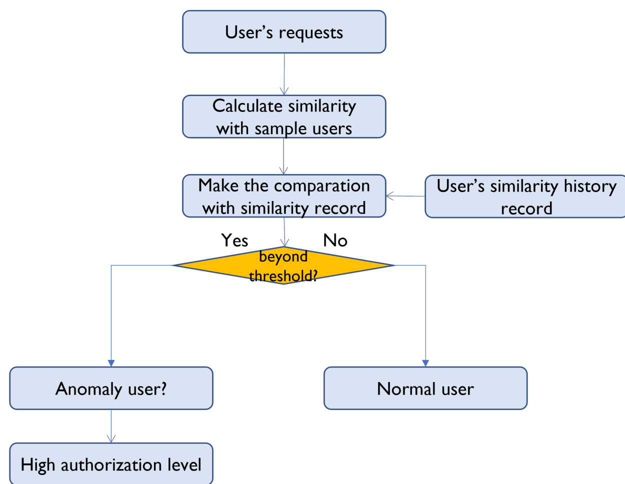


Figure 5.2: The main steps to check if the current user is abnormal.

Popular recommendation algorithms such as the *user-based collaborative filtering* algorithm (UBCF) [20] offers a practical cosine similarity calculation method. Initially, the recommendation module completes a similarity comparison between each user and all the valid users in the system. In this study, we adopt the similarity comparison scheme inspired from UBCF. The DDoS detection system carries out four key steps to consolidate the multiple similarity values into an anomaly detection procedure, where each online user is assessed

by the similarity manager amid the routine access process. Given an online user u_x, we denote its requests as list $L_x = \{o_{x_0}, o_{x_1}, \ldots, o_{x_i}\}$. Thus, we express the access of data item o_i from user u_x as a step function

$$r_{xi} = \begin{cases} 1, & \text{if } o_i \in L_x \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

In our similarity manager, we choose to apply the *pearson correlation coefficient* algorithm to compute similarities between any pair of two users u_x and u_k. Specifically, with respect to user u_x, the similarity is calculated by the accesses comparison defined in (5.1). Hence, the similarity between users u_x and u_k is formally expressed as

$$sim(u_x, u_k) = \frac{\sum_{i \in O}(r_{xi} - \bar{r}_x)(r_{ki} - \bar{r}_k)}{\sqrt{\sum_{i \in O}(r_{xi} - \bar{r}_x)^2}\sqrt{\sum_{i \in O}(r_{ki} - \bar{r}_k)^2}}. \tag{5.2}$$

where $\bar{r}_x$ and $\bar{r}_k$ are the average access record numbers for users u_x and u_k. It is worth noting that the average numbers are float numbers in a window between 0 and 1 derived from r_{xi} and r_{ki} lists. O is the entire data item list in the cloud storage system.

The data structure of each online user in the similarity manager is an array of user-similarity pairs, which is defined in (5.3). Given user u_x and the corresponding similarity value $sim(u_x, u_k)$ with respect to user u_k, we express the user-similarity pair ϕ_{xk} as

$$\phi_{xk} =< u_k, sim_(u_x, u_k) > . \tag{5.3}$$

Due to high diversity and high overhead on computing similarity among all legitimate users, we propose a user sampling module at the heart of the DDoS detection system to minimize the overhead of similarity calculation while improving detection accuracy. In the user sampling module, we designate a $fitness$ value to select candidate users from the legal

user set. Formally, fitness f_k quantifies the degree at which user k is an appropriate sample user. We derive f_k from (5.4) as:

$$f_k = -\frac{n_O}{n_{O_k}} \sum_{i \in R_k} (r_{ki} - \bar{r}_i)^2. \tag{5.4}$$

where n_O is the number of available data items in the whole cloud storage and n_{O_k} is the number of valid data items requested in the history record of user k. If a user has a long access history where a wide variety of data items were accessed, then this user tends to have a high fitness. The summation in (5.4) resembles a variance estimation, where R_k is a set of data objects rated by users k, r_{ki} is a rating record in the history of data item o_i by user k and $\bar{r}_i$ is the average rating by all the legal users. It is evident that a low variance modeled by (5.4) signifies that a user is an appropriate representative of legal users to be included in the user sample set. Therefore, we set a the fitness to a negative value to acquire a high fitness value in the user sampling module.

The similarity manager calculates similarity values of user u_x with respect to all the users registered in sample user set U. Therefore, given user x, there is a list of $sim(u_x, u_k)$, where we have $k \in S$. In the DDoS detection procedure, anomalies are identified by tracking a changing trend in similarity value ϕ_{xk}. We assume that ϕ_{xk} and ϕ'_{xk} are the current and previous similarity values between users u_x and u_k. We denote δ_{xk} as the discrepancy between ϕ_{xk} and ϕ'_{xk}. Thus, we have $\delta_{xk} = |\phi_{xk}$ - $\phi'_{xk}|$.

Let f_x be the status of the xth user to be evaluated. $f_x = 0$ signifies that u_x is a legitimate user; $f_x = 1$ indicates that u_x is an anomalous one. If δ_{xk} measuring a shift between the current and previous similarity values exceeds a prescribed threshold α, the detection procedure will mark the user as abnormal. More formally, we have

$$f_x = \begin{cases} 1, & \text{if } \delta_{xk} \geq \alpha \\ 0, & \text{otherwise.} \end{cases} \tag{5.5}$$

Likewise, anomalies are pinpointed by monitoring similarity value $sim(u_x, u_k)$. In case $sim(u_x, u_k)$ goes beyond a designated threshold β, user u_x is flagged as an anomalous one. Thus, we express this rule as

$$f_x = \begin{cases} 1, & \text{if } sim(u_x, u_k) \geq \beta \\ 0, & \text{otherwise.} \end{cases} \tag{5.6}$$

Merging (5.5) and (5.6), the following consolidated rule is adopted to discover anomalies.

$$f_x = \begin{cases} 1, & \text{if } \delta_{xk} \geq \alpha \text{ or } sim(u_x, u_k) \geq \beta \\ 0, & \text{otherwise.} \end{cases} \tag{5.7}$$

In the nutshell, we summarize the DDoS detection procedure into the following four steps, which entail *sample user selection*, *online user request collection*, *similarity calculation*, *similarity comparison*, respectively.

- *Sample User Selection:* The DDoS detection system groups a set of sample users by the sampling technique. The sample users are applied for making similarity comparisons with online users.

- *Online User Request Collection:* While the cloud server acquires a request list from online users, the detection system gleans online requests as lists for all the users.

- *Similarity Calculation:* The DDoS detection system dynamically and repeatedly calculates the similarity measures for all the users. The similarity result of the kth user is expressed as a list of user-similarity pairs, which contains $(u_{k0}, sim(u_x, u_{k0}))$, $(u_{k1}, sim(u_x, u_{k1}))$, ... , $(u_{kn}, sim(u_x, u_{kn}))$.

- *Similarity Comparison:* The user-similarity list of each user is compared against its historical record. If δ_{xk} and $sim(u_x, u_k)$ are moving beyond the two designated thresholds α and β, the DDoS detection model identifies the user as abnormal.

5.3 Algorithm Design

Our detection model generates a sample user set prior to kicking off the detection procedure, which periodically keeps tracks of legitimate users' behavioral trends. We illustrate below the user sampling process in Algorithm 5.

Algorithm 5: The sample user selection algorithm.

Input:
U': All the legitimate users in a cloud storage system;
O: Data item set contains valid ratings in a history record;
Output:
U: Sample user set
1: $avg(O)$;
2: **for** all $u_{x'} \in U'$ **do**
3: $sampleUserSet.add(fitness(u_{x'}, avg))$;
4: **end for**
5: $sampleSelection(U, h)$;
6: **return** sample user set U;

In Step 1, we calculate the average ratings of all the data items accessed by all the legal users. The *sampleUserSet()* function generates a list of user-fitness pairs governed by Equation (5.4) in Step 3. In case that the fitness values of all the users are obtained, the system elects the top h user as sample users.

When the cloud storage server is accessed by online users, our detection system assumes that every user may become a potential attacker. As such, each online user is thoroughly measured and speculated by our detection system while handling and processing user requests.

Algorithm 6 depicts the procedure of a similarity- based detection model to guard against application-layer DDoS attacks in cloud storage. Recall that a total of four steps (see also Section 5.2.3) are carried out to detect anomalous users by comparing the similarities between each current online user against all the sample users originated by Algorithm 5.

The input information of Algorithm 6 embraces a given user u_x accompanied by a list of online requests O_x, a sample user set U, the similarity history record $S(u_x)$ of user u_x and the two thresholds α, β.

Algorithm 6: The high-level controller of anomaly user detection.

Input:
A given current user ID u_x; /* potential attacker */
The online requests O_x of u_x;
Sample user set U;
The similarity history record $S(u_x)$ of u_x;
The threshold settings α and β;
Output:
f_x: the status of user u_x;
1: $C(u_x) = null$;
2: **for** all $u_k \in U$ **do**
3: $C(u_x)$.add(SimilarityCalculation(u_x,u_k));
4: **end for**
5: **for** all $s(u_x) \in S(u_x)$ **do**
6: **for** all $c(u_x) \in C(u_x)$ **do**
7: **if** $s(u_x).getKey() == c(u_x).getKey()$ **then**
8: $\delta_x = |s(u_x).getValue()$ - $c(u_x).getValue()$
9: **if** $\delta_x >= \alpha$ **then**
10: $g_x = g_x + 1$;
11: **end if**
12: **end if**
13: **end for**
14: **end for**
15: **if** $g_x >= \beta$ **then**
16: $f_x = 1$; /* Anomalous User */
17: **else**
18: $f_x = 0$; /* Legitimate User */
19: **end if**
20: **return** f_x;

In Algorithm 6, Steps 2-4 repeatedly undertake similarity calculations to construct a list of similarity values between the current user u_x and sample users in set U forged by Algorithm 5. More specifically, the similarity calculation is implemented by function $SimilarityCalculation()$ in Step 3.

Steps 5-16 are responsible for comparing between the current similarities calculated in Step 3 and those registered in history record $S(u_x)$, followed by checking if the user is an anomalous user. Set $S(u_x)$ represents a history similarity record between current user u_x and those elected in sample user set U. In Step 7-8, we gauge similarity discrepancy between the similarity values extracted from $S(u_x)$ and $C(u_x)$. The $getKey()$ and $getValue()$ functions acquire user IDs and similarity values from the user-similarity pairs (see also Section 5.2.3). The similarity comparison is implemented in Steps 10 and 16. While a single similarity value changes beyond the threshold α, the abnormal value increased by 1 to record the similarity anomaly change. While the number of anomaly changes beyond the point β, the detection system identifies the current user u_x as an abnormal user and return the user Id at Step 18.

In our design, the user sampling module ought be performed in Algorithm 5 prior to the DDoS detection procedure. This order of execution is expected because sample user generations are judiciously maintained by cloud storage regardless of the DDoS detection procedure.

5.3.1 Overhead Analysis

We reckon that the similarity-based DDoS detection system imposes a relatively low computation overhead. Compared to general machine-learning- algorithm-based detection models, our proposed system has a leading edge in terms of minimizing the cost of data training and analysis. In the worst scenario, the process of electing sample users consumes computing resources in the case of a massive number of legal users. To cut down the overhead of selecting sample users, we implement a light-weight user sampling module by incorporating the random selection algorithm, which randomly picks a relatively small group of legitimate users as the candidates without hefty cost.

5.4 Performance Evaluation

The main design goal of the similarity-based DDoS detection is, as we emphasized early, to reduce the overhead with the competitive detection accuracy. We emulate a storage system that maintains a collection O of data items, which are concurrently accessed by a number of clients. The experiment selects a request object sequence of HTTP requests in the server log as training data and HTTP request sequences as test data, whereas data of DDoS attacks are not included. To verify the similarity-based DDoS detection mechanism's ability of detecting attacks, DDoS attacks anchored on HTTP requests are simulated. We suppose that 20 attacking nodes send GET requests to a target a web server to implement orchestrated DDoS attacks.

5.4.1 Evaluation of Overhead

This section is focused on the analysis of the overhead incurred in the DDoS detection procedure compared against two existing anomaly-based DDoS detection techniques. The storage system automatically triggers a series of requests to the server while an active user is accessing the system. Therefore, the system must filter out other behaviors and extracts the keywords in the request sequence according to the method of preprocessing data in the database abnormality detection system, where the accesses are arranged into a sequence in a chronological order.

In the experiment, we elect a request object sequence of 1000, 5000, and 10000 HTTP requests from the server log as training data. Next, we measure I/O cost among the anomaly-based detection techniques powered by multiple machine learning algorithms, in which the size of training set is varied. The experiment results indicate that our similarity-based detection model leads the lowest overhead on pre-detection compared with the two alternatives - the behavior-learning-based algorithm and the clustering-based algorithm.

Fig. 5.3 illustrates that with an increasing number of users' history behavior in the training set, the overhead of training costs among the three modeling algorithms surge

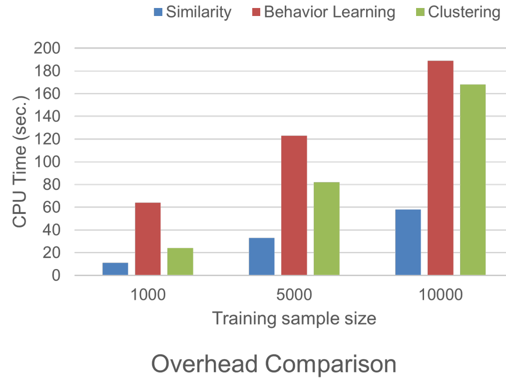


Figure 5.3: The time cost vs. an increasing size of the training set (Users History Behavior).

- an expected trend speculated before conducting the experiment. Our similarity-based detection model, nevertheless, has a leading edge against the other two solutions in terms of modeling overhead: the overhead incurred by our model is less sensitive to the size of training datasets. It is arguably true that the similarity calculation and comparison exhibit the lowest complexity compared with the user's behavior features extraction module and the classification module. Similarity calculation - the main contributor to similarity-based detection - has its time complexity capped at $O(n^2)$: such a low complexity makes our design a winner across the board among various machine learning-based DDoS detection techniques. Furthermore, our proposed system design gauges the similarities only between target users and a sample user set, thereby immensely curtailing the overhead of training cost.

5.4.2 Evaluation of Accuracy

In this group of experiments, we choose the request object sequence of 5000 HTTP requests as a test data, and we elect 1000 requests to build a the training set. The DDoS attack is emulated as follows: (1) a total of 50 attacking nodes deliver GET requests to the target Web server to launch a DDoS attack; (2) the content of the attack requests are randomly generated as high-frequency GET requests.

The results plotted in Fig. 5.4 reveal that with an increasing number of accessed data objects, the detection accuracy of the three models floats from 78% to 95%. Theoretically

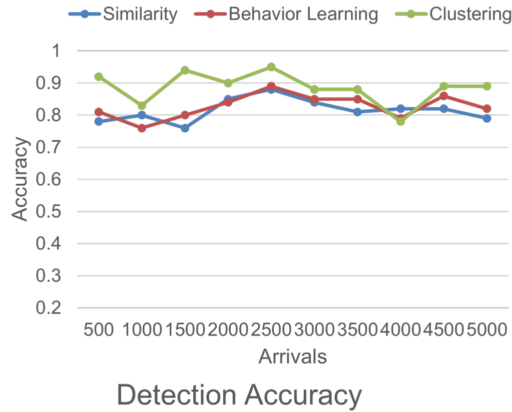


Figure 5.4: Detection Accuracy with the an increasing number of data items.

speaking, it is strenuous to push the maximum accuracy all the way to 100% regardless of the deployed modeling algorithms: obtaining false negatives for anomaly analysis becomes unavoidable. Obviously, the clustering-based detection technique renders a relatively high accuracy on DDoS detection, but the overall accuracy performance of the three models does not gap away from each other.

5.5 Summary

We started this chapter with an overview of DDoS attacks as well as detection techniques in cloud storage systems. After presenting a broad list of techniques for DDoS detection, we placed a focal point to the DDoS threats and detection techniques at the application layer. The core component of our study is centered around anomaly detection techniques as a safeguard against application-level DDoS threats. We delved into multiple innovative and promising ways of applying machine learning solutions to efficiently identify DDoS zombies in cloud computing environments. Next, we shed bright light on the challenges in DDoS detection, followed by proposing a similarity-based DDoS detection system. The overarching goal of the designed system is to trace similarity changes among users to diagnose the behaviors of users who are out of ordinary. Such users will be marked by our system as potential DDoS attackers at the application level.

At the heart of our proposed DDoS detection system, we devised a user sampling module and a similarity comparison module. The sampling module attempts to elect the most appropriate user set from existing legal users, whose history access records are thoroughly inspected. The implementation of the user sampling module is crucial and indispensable to the detection system because this module is forged to (1) minimize the overhead of DDoS attack detection and (2) to curtail the effects of reasonable changes from a small group of users (e.g., updates of users' interests). Along this line, the similarity comparison module cuts down on the calculation cost spent in comparing among users behaviors. We expect that highly accurate DDoS threats detection will be successfully implemented in a light-weighted manner by the virtue of our proposed users' behavior analysis.

Chapter 6

Popularity-Aware Malware Detection

Cloud storage systems facilitate a platform to store and manage cyberspace data for a wide range of applications. In cloud storage connected to the internet, a large amount of data is uploaded and accessed by numerous users. Thus, security and privacy of data is of utmost importance to end users regardless of the nature of the data being stored in clouds. After outlining the development of cutting-edge of cloud storage systems, we elaborate data security issues in cloud storage. We pay particular attention to malware detection techniques customized for cloud storage. The overarching goal of our solution articulated in this chapter is to guarantee that data are malware free in cloud storage prior to being accessed by end users. Inspired by the architecture of cloud storage, we propose a popularity-aware malware detection strategy to enhance the security of cloud storage systems by protecting high-risk data. Data risk is gauged through popularity, because popular data deserve high priority when it comes to access frequencies. Our designed technique speculates data popularity, which is an avenue to prioritize data objects amid time-consuming malware detection procedures. Our technique is conducive to keeping malware at bay when popular data are frequently accessed by clients.

In Section 6.1, we start the research road map by presenting a basic idea for the development of popularity-aware malware detection techniques. We depict a high-level system architecture for schedulers supporting popularity-aware malware detection in cloud computing platforms in Section 6.2. Then, we elaborate the concepts and main steps in Section 6.3. Finally, in Section 6.4 we outline the scheduling algorithms that optimize the performance of malware detection systems. We summarize this chapter in Section 6.5.

6.1 Basic Idea

It is arguably true that cloud computing platforms can be jointly optimized by incorporating multiple dimensions like connection efficiency, access security, data placement, and scheduling. In this part of the dissertation study, we pay heed to security issues centered around cloud storage systems. The evidence from the prior studies (see, for example, [69] and [27]) shows that they cannot unify the service because of the untrusted remote machines. In addition, any security solution customized for cloud storage is required to impose a negligible performance overhead. In our pilot study, we aim to demonstrate that scheduling techniques are capable of enhancing the performance of malware detection deployed in modern cloud storage.

The basic idea of our solution is to schedule a sequence of data objects in which malware are detected. Scheduling decisions should be made in a way that high-risk data are scanned by a malware detector in an early phase followed by low-risk data. When we set the high risk data object prior to low risk data on malware detection sequence, data security will be improved because the high-risk data experience a high intrusion possibility to systems and a high infection possibility to users.

One of the vital factors affecting the probability of successful intrusions is the number of access points [22]. As such, we advocate for gauging the risk of data objects using popularity measures. Compared with non-popular data, popular data objects have a high access frequency from active users. Popular data are treated as high-risk data because of the following two reasons.

- First, malware infection in popular data becomes a serious threat for enormous number of users. Each popular data object is retrieved by a large group of users, who will be victimized the malware codes.

- Second, there is a strong likelihood for popular data to be frequently accessed in a short time period. The malware detection system ought to ensure that the popular data are malware-free before being accessed by users.

For the above reasons, scheduling a detection sequence among data objects according to access popularity improves the data security of cloud storage systems by detecting malware of high-risk data in the first place.

6.2 System Architecture

Fig. 6.1 depicts the system architecture of our proposed malware detection scheduling system, where a malware detection manager collaborates with a scheduler in clouds. The entire system is responsible for scheduling a detection list in which data objects are scanned for malware before being actively accessed by users. During the online malware detection procedure, the scheduler makes judicious decisions on a detection order, in which popular data are assigned high priorities to mitigate malware threats.

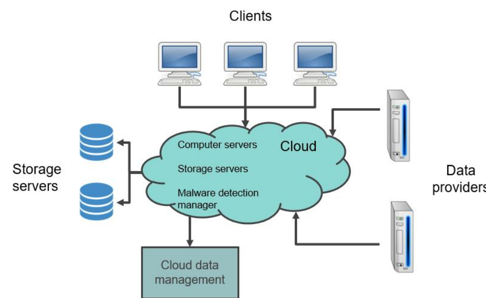


Figure 6.1: The architecture of a malware detection scheduler running in clouds. Popular data objects receive a high priority to be scanned earlier than unpopular data objects.

In cloud computing environments, data are distributed across multiple storage nodes in data centers. Unsurprisingly, it is straightforward to incorporate prevalent malware detection techniques into cloud computing systems. For example, the leading-edge malware detection systems reported in [10] and [31] can be applied to fight malware in cloud storage services.

The overarching goal of the system is to optimize malware-detection performance by alleviating malware risks. To achieve this design goal, a management module is engaged

to gauge data popularity measures, which are maintained as key components in metadata. Popular data objects are likely to be accessed by a growing number of users within in a short time period. It is arguably true that the scheduler in this architecture is adroit at governing the malware detection procedure, where detection priorities are configured in accordance to monitored data popularities. With the scheduler in place, popular data objects are handled by the detection module earlier than unpopular counterparts. Such scheduling decisions play a vital role in speeding up malware detection performance, because the time spent in identifying malware can be conserved by postponing the detection of unpopular data that impose low risks in clouds.

6.3 Concepts and Main Steps

To assess the popularity of data objects, we adopt a recommendation scheme in the malware-detection scheduling module. The recommendation scheme predicts a list of recommended data objects for each user who is actively accessing data from cloud storage. The malware-detection manager merges individual recommendation lists of multiple users into a single scheduling list for the malware detector in our system. Specifically, the manager carries out the four steps to consolidate multiple recommendation lists into a detection list, where the most popular data are scanned by the malware detector in an early stage. The data structure of detection lists is an array of object-weight pairs, which is defined below. Given object o_i and its weight $w_{x,i}$ with respect to user u_x, we express object-weight pair $\phi_{x,i}$ as:

$$\phi_{x,i} =< o_i, w_{x,i} > . \tag{6.1}$$

Table I summarizes a list of recommendation algorithms that can be plugged into the malware-detection scheduler. The popularity of data objects can be assessed by one of the following recommendation algorithms in Table I. Each recommendation algorithm has

recommendation type	basic idea	advantages	common algorithms
Content-based recommendation [90]	recommend an item to a user based on a description of the item and a profile of the user's interests	small number of structured attributes, simplicity, understandability	decision tree
Collaborative Filtering Recommendation [133] [104]	use the existing user's past behavior or comments to provide the product which conforms with the current user's requirements	good performance on large number of user and items	user based collaborative filtering item based collaborative filtering
Knowledge-based Recommendation [17] [36]	Systems that rely on knowledge sources of user requirements and domain knowledge	rely on knowledge sources that were not being employed by the more widely-used techniques	case-based recommendation constraint-based recommendation
Hybrid recommendation [18]	a combination of recommendation components or logic	high accuracy, easy to implement	Feature combination, weighted, switching

Table I: A list of candidate recommendation algorithms are readily plugged into the popularity-aware malware detection scheduler.

its unique advantages, depending on workload conditions such as number of active users, number of data objects, and the performance of cloud storage. As a case study, we import user based collaborative filtering as an underpinning technique to implement the popularity-aware scheduler.

Let us introduce the fundamental concepts of *key*, *key-value pairs*, *weight*, and *blocks* before diving into the description of the following four steps. We define data objects' identifiers (IDs) as *keys*, meaning that any data object can be readily referenced through its key. A data object is organized in the data structure of a *key-value* pair, where value is the content of the data object. We refer to the access frequency of a data object as a weight - an importance feature to capture the popularity of the data object. In a cloud storage system, data objects are basic storage units, which form large chunks called *blocks*. In other words, a data block is comprised of a group of data objects; all the data blocks share a fixed size. It is noteworthy that the block size can be configured by in the cloud storage, in which the default block size of our system is 64 MB.

- To retrieve data objects and the corresponding weights in a single user recommendation list to calculate the number of occurrences and weight for each key.
- Data blocks and data objects entail a two-layer data organization, where each data object belongs to a parent data block. The second step is to map the data objects' keys to their data blocks in the cloud storage.
- To calculate the summation of weights of each data block so that a detection list is constructed to embrace to-be-scanned data blocks accompanied by the corresponding

weights. The weight of each data block indicates the block's popularity, which measures the future access frequency of the block.

- To schedule the items in the detection list according to the decreasing values of weights associated to the data blocks. In this step, data blocks with high weights are treated as popular data that are likely to be accessed by a large group of users in the not-too-distant future.

6.4 Algorithms

Algorithm 7 depicts the procedure of a popularity-based malware detection system in cloud storage, in which the above four steps (see Section 6.3) are carried out to schedule a detection list by merging all the recommendation lists predicted for active users in clouds.

The input information of Algorithm 7 includes user I/O access history and ratings as well as a set of data blocks in a cloud storage, which is formally defined as $B = \{b_1, b_2, ..., b_n\}$. Here, we assume the total number of blocks managed in the cloud storage is n.

The output of Algorithm 7 is a scheduled detection list *detList*, which contains an array of *block-weight* pairs. Ideally, the length of scheduled list *detList* is identical to the size of set B. Thus, we have

$$detList.size() = |B| = n. \tag{6.2}$$

If system administrators opt for cutting back the overhead spent in diagnosing a large number of blocks, a subset of set B will be elected to originate scheduled list *detList*. Intuitively, increasing the length of list *detList* can substantially raise the overhead of scheduling data blocks and detecting malware. List *detList*'s length ought to be appropriately chosen based on the system utilization and workload of the cloud storage.

Given data block b_k and its weight w_k, we define $\alpha_k = (b_k, w_k)$ as a block-weight pair for the kth block b_k. The detection list - an output of Algorithm 7 - is formally expressed

as Eq. 6.3, where all the block-weight pairs are scheduled by the Algorithm in a decreasing order of the weights in the block-weight pairs.

$$detList = \{\alpha_1, \alpha_2, ..., \alpha_n\} = \{(b_1, w_1), ..., (b_n, w_n)\},$$
$$\text{where } w_1 \geq w_2 \geq ... \geq w_n. \quad (6.3)$$

It is noteworthy that the weight of a block captures the popularity of data objects residing in the block. The most popular blocks are scheduled to be detected at the beginning of *detList*; the least popular ones are postponed toward the end of the list.

Algorithm 7: The high-level controller of malware detection.

Input:
User I/O access history and rating records;
$B = \{b_1, b_2, ..., b_n\}$; /* data blocks to be detected */

Output:
$DetList = \{\alpha_1, \alpha_2, ..., \alpha_n\}$; /* A detection list */

1: **for** all $u_x \in U$ **do**
2: $P(u_x)$ = UbasedCoFiltering(u_x);
3: **end for**
4: **for** all $\alpha_k \in DetList$ /* Initialize $DetList$ */ **do**
5: SetWeight(α_k, 0); /* Initialize the weight of α_k */
6: **end for**
7: **for** all $u_x \in U'$ **do**
8: **for** all $\phi_{x,i} \in P(u_x)$ **do**
9: o_i = GetDataObject($\phi_{x,i}$);
10: $w_{x,i}$ = GetWeight($\phi_{x,i}$);
11: b_k = GetBlock(o_i, B);
12: detList[w_k]+= $w_{x,i}$;
13: **end for**
14: **end for**
15: Sort(detList[w_k]);
16: **return** *detList*;

In Algorithm 7, Steps 1-3 repeatedly carry out user-based collaborative filtering to construct recommendation lists for the users in set U. More specifically, the collaborative filtering strategy is implemented by function $UbasedCoFiltering()$ in Step 2.

In our design, user-based collaborative filtering ought be performed in *Step 2* prior to establishing recommendation lists. This order is expected, because recommendation lists (see *Step 2*) are judiciously maintained by cloud storage regardless of the malware detection procedure. In a real-world cloud, the recommendation lists are proactively updated while data objects are being accessed by users.

Steps 4-6 control the initialization of the weights of the block-weight pairs in *detList*; the initial value of the weights is 0. Let U' represent a set of users who are actively accessing cloud storage amid the malware detection process. Because user information is retained in user set U, U' is a subset of U (i.e., $U' \subseteq U$). *Steps 7-14* repeatedly calculate each user's weights with respect to data objects. In particular, *Step 9* derives data object o_i from object-weight pair $\phi_{x,i}$ (see also the $GetDataObject()$ function). *Steps 10* and *11* obtain weight $w_{x,i}$ and block b_k from object o_i (see also the $GetWeight()$ and $GetBlock()$ functions). In *Step 12*, weight w_k is modified by augmenting intermediate result $w_{x,i}$ yielded from *Step 10*.

Finally, Step 15 sorts block-weight pairs in a non-increasing order of weights in detection list *detList*. Such a detection schedule is made by the $sort()$ function in the algorithm.

We reckon that the malware detection scheduler imposes relatively low computation overhead, because recommendation lists normally are originated prior to the malware detection procedure. In the worst case scenario, the lack of user requests (e.g., $U' = \emptyset$) makes it strenuous build a recommendation list to project data popularity. In this case, the system gracefully downgrades to offline malware detection, in which a detection schedule is no longer needed. In such an offline malware detection case, *Steps 7-14* In Algorithm 7 will be excluded.

6.5 Summary

We started this chapter with an overview of cloud storage systems. After presenting a list of emerging techniques adopted in cloud storage, we discussed cloud security issues to be addressed in data storage systems in clouds. The focus of this last piece of the dissertation

study is centered around malware detection in cloud storage. We illustrated multiple ways of applying machine learning solutions to efficiently identify malware codes in clouds. Next, we shed bright light on the challenges confronted in malware detection in the realm of cloud storage. To overcome these challenges, we proposed a popularity-aware malware detection system, which schedules a malware- detection sequence in a way that high-risk data are detected prior to low- risk counterparts in clouds.

At the heart of our proposed malware detection system, we designed a user-based collaborative filtering module to predict data popularity using established recommendation lists. We delineated the popularity-aware algorithm to (1) prioritize data blocks and (2) make detection schedules in a way to enhance the security of cloud storage systems. Along this line, we expect that a diversity of machine learning techniques can be employed to forecast data popularity, which in turn can determine malware-detection schedules. It is intriguing to quantitatively compare a handful of prediction solutions to figure out which one delivers the best performance for the malware detection system in cloud computing environments, where big data must be constantly scanned.

Chapter 7

Conclusions and A Future Research Plan

In previous chapters, we propose a recommendation algorithm based popularity calculation technique. In this chapter, we introduce the proposed practice design of popularity-aware data reconstruction, cache replacement strategy, DDoS detection and malware detection technology.

This chapter consists of three sections, namely, concluding remarks presented in Section 7.1, a future research plan moving beyond this dissertation study (see Section 7.3), and a summary of the chapter in Section 7.4.

7.1 Conclusions

7.1.1 Data processing and scheduling

Thanks to online services, cloud computing systems are widely used to support a variety of application domains. Active users may access cloud-based applications at any time from anywhere through the Internet. There is a dire demand to embark on time-consuming malware detection procedures while users are accessing data from the clouds. There are three concurrent research paths toward tackling this challenge.

- First, resources must be fittingly partitioned and allocated among users, systems, and malware-detection services. Making a good tradeoff is the fundamental key in resource management for future malware detection systems in clouds. On the one hand, if the majority of resources are dedicated to malware detection, user experience will be dragged down. On the other hand, if we favor user response time by limiting resources for malware detection, data are likely to be accessed without malware diagnosis.

- Second, it is difficult, if not futile, to scan a massive amount of data to detect malware in big data arenas. This issue can be addressed by either detecting subsets of the big data or lowering the detection frequency. Data selection algorithms should be developed to pick security-sensitive data from non-sensitive ones. A malware detection system ought to guarantee that high-risk data are diagnosed before being accessed by users. Frequency selection algorithms should be in charge of determining the most appropriate interval between two consecutive detection instances.

Data processing and scheduling are among the dominant methods to optimize the efficiency of malware detection systems deployed in a wide range of online cloud services. In what follows, we shed bright light on novel data processing and scheduling techniques aiming to improve system execution efficiency. The proposed models for data processing, managing and analysis of data popularity. The models contain a flow of streaming data transferred and processed in the two modules, namely, (1) prioritize data blocks and (2) make detection schedules.

7.2 Advanced algorithm exploration

We will seamlessly integrate the recommendation module with a pre-fetching mechanism to overcome this obstacle. We will pay heed to reduce overhead and push up pre-fetching accuracy of the data pre-fetching mechanism by applying the recommendation algorithms. We plan to leverage the recommendation list for each active user to harvest a popular data list cultivating data pre-fetching. A second research issue that is worth being explored in the future is to integrate advanced recommendation algorithms into PDC. The recommendation module implemented in the current version is unable to address challenging issues like frequent user-interest shifts and temporal dynamics. We plan to extend the PDC implementation by incorporating feedback-based recommendation algorithms like [67]. Feedback on recommendation results will be utilized to capture dynamic and dramatic changes in user interests.

7.3 A Future Research Plan

In the foreseeable future, we will push forward the following research agenda beyond this dissertation study. The research plan is comprised of the following two directions.

7.3.1 Research Direction 1

Data mining in big data storage systems. I am thrilled to continue my dissertation work on exploiting cutting-edge machine learning and data mining algorithms to improve efficiency, reliability, and scalability of big data storage systems. I will be engaged in devising new models and methodologies to tackle various data mining challenges. My recommendation algorithm-based popularity scheduler will be extended to revamp data prefetching performance in large-scale storage systems. I intend to propose new recommendation algorithms catering to data prefetching mechanism by virtue of accurate predictions of future accesses. Prefetching data that users may not eventually request inevitably waste storage, network bandwidth, as well as scarce cache capacity in big data storage systems. As such, I have a concrete plan to seamlessly integrate a recommendation module with a prefetching mechanism to overcome this obstacle. A second research task that is worth being explored along this direction is to incorporate advanced recommendation algorithms. The recommendation module implemented in my current prototype system has an open architecture in the way that new algorithms can be readily plugged in. The expanded algorithms will be focused on addressing challenges like frequent user-interest shifts and temporal dynamics.

7.3.2 Research Direction 2

Incremental computing for big data using machine learning. I will solve big data problems through incremental computing techniques powered by machine learning and data mining. Periodic data analytic jobs running in storage systems tend to lead to skews in computing and I/O resources. As a second future research direction, I will explore incremental computing techniques to judiciously handle periodic big-data jobs. I will propose

an incremental computing module to alleviate I/O resource skewness. Again, my incremental computing module will be governed and optimized by machine learning algorithms that predict future data access patterns of periodic analytic jobs. The incremental computing module aims to partition each big job into an array of small tasks that incrementally process data archived on system systems.

7.4 Summary

In this dissertation, we focused on the popularity-aware applications for online big data servers. We proposed a popularity calculator for popularity driven data processing as well as four applications. The experiment results confirm that our proposed models and system design are capable of achieving optimized performance in various perspectives. We proudly summarize the key contributions of the dissertation research in the four bulleted items.

- First, we developed an erasure-coded storage system called POST, which seamlessly integrates the efficient data archival and online reconstruction techniques. We implemented a k-prototype clustering controller to archive unpopular data that attract a limited number of accesses. Our POST system is reliant on the clustering controller to group files into multiple clusters, in each of which files share similar features.

- Second, we elaborated the design and implementation of a web-proxy-cache replacement system called *PDC*. We built a recommendation module to harvest popularity values of data objects in storage systems. Our PDC is reliant on the recommendation system to gauge data popularity, thereby evicting unpopular data from caching system.

- Third, we devised a user sampling module and a similarity comparison module. The sampling module attempts to elect the most appropriate user set from existing legal users, whose history access records are thoroughly inspected. The implementation of the user sampling module is crucial and indispensable to the detection system because this module is forged to (1) minimize the overhead of DDoS attack detection and (2)

to curtail the side effects of reasonable changes from a small group of users due to a diversity of reasons such as updated user interests.

- Fourth, we designed a user-based collaborative filtering module to predict data popularity using established recommendation lists. We delineated the popularity-aware algorithm to (1) prioritize data blocks and (2) make detection schedules in a way to enhance the security of cloud storage systems.

Bibliography

[1] Big data and what it means:. https://www.uschamberfoundation.org/bhq/big-data-and-what-it-means.

[2] The books dataset. https://www.kaggle.com/zygmunt/goodbooks-10k#books.csv.

[3] The movies dataset. https://www.kaggle.com/rounakbanik/the-movies-dataset#movies_metadata.csv.

[4] M. Abdelsalam, R. Krishnan, Y. Huang, and R. Sandhu. Malware detection in cloud infrastructures using convolutional neural networks. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 162–169. IEEE, 2018.

[5] M. Abdelsalam, R. Krishnan, and R. Sandhu. Online malware detection in cloud auto-scaling systems using shallow convolutional neural networks. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 381–397. Springer, 2019.

[6] U. Acharjee. *Personalized and artificial intelligence Web caching and prefetching*. PhD thesis, University of Ottawa (Canada), 2006.

[7] J. Alghazo, A. Akaaboune, and N. Botros. Sf-lru cache replacement algorithm. In *Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004.*, pages 19–24. IEEE, 2004.

[8] W. Ali. Performance improvement of web proxy cache replacement using intelligent greedy-dual approaches. *Performance Improvement*, 9(8), 2018.

[9] S. Alqahtani and R. F. Gamble. Ddos attacks in service clouds. In *2015 48th Hawaii International Conference on System Sciences*, pages 5331–5340. IEEE, 2015.

[10] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258, 2011.

[11] Ş. Ş. Arslan, B. Parrein, and N. Normand. Mojette transform based ldpc erasure correction codes for distributed storage systems. In *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4. IEEE, 2017.

[12] B. Athiwaratkun and J. W. Stokes. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2482–2486. IEEE, 2017.

[13] A. Bakshi and Y. B. Dujodwala. Securing cloud from ddos attacks using intrusion detection system in virtual machine. In *2010 Second International Conference on Communication Software and Networks*, pages 260–264. IEEE, 2010.

[14] K. S. Bhosale, M. Nenova, and G. Iliev. The distributed denial of service attacks (ddos) prevention mechanisms on application layer. In *2017 13th International Conference on Advanced Technologies, Systems and Services in Telecommunications (TELSIKS)*, pages 136–139. IEEE, 2017.

[15] D. Borthakur et al. Hdfs architecture guide. *Hadoop Apache Project*, 53(1-13):2, 2008.

[16] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, et al. Web caching and zipf-like distributions: Evidence and implications. In *Ieee Infocom*, volume 1, pages 126–134. INSTITUTE OF ELECTRICAL ENGINEERS INC (IEEE), 1999.

[17] R. Burke. Knowledge-based recommender systems. *Encyclopedia of library and information systems*, 69(Supplement 32):175–186, 2000.

[18] R. Burke. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370, 2002.

[19] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011.

[20] T. Cao, X. Peng, C. Zhang, T. K. Al Tekreeti, J. Mao, X. Qin, and J. Huang. A popularity-aware reconstruction technique in erasure-coded storage systems. *Journal of Parallel and Distributed Computing*, 146:122–138, 2020.

[21] Ö. Cepheli, S. Büyükçorak, and G. Karabulut Kurt. Hybrid intrusion detection system for ddos attacks. *Journal of Electrical and Computer Engineering*, 2016, 2016.

[22] K. Chatterjee, V. Padmini, and S. Khaparde. Review of cyber attacks on power system operations. In *2017 IEEE Region 10 Symposium (TENSYMP)*, pages 1–6. IEEE, 2017.

[23] Y. Chen, Y. Zhou, S. Taneja, X. Qin, and J. Huang. ahdfs: an erasure-coded data archival system for hadoop clusters. *IEEE Transactions on Parallel and Distributed Systems*, 28(11):3060–3073, 2017.

[24] Z. Chen, G. Xu, V. Mahalingam, L. Ge, J. Nguyen, W. Yu, and C. Lu. A cloud computing based network monitoring and threat detection system for critical infrastructures. *Big Data Research*, 3:10–23, 2016.

[25] J. Cheng, M. Li, X. Tang, V. S. Sheng, Y. Liu, and W. Guo. Flow correlation degree optimization driven random forest for detecting ddos attacks in cloud computing. *Security and Communication Networks*, 2018, 2018.

[26] I. CLEVERSAFE. Cleversafe dispersed storage. *Open source code distribution: http://www. cleversafe. org/downloads*, 2008.

[27] S. Contiu, S. Vaucher, R. Pires, M. Pasin, P. Felber, and L. Réveillère. Anonymous and confidential file sharing over untrusted clouds. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, pages 21–2110. IEEE, 2019.

[28] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE, 2013.

[29] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559–570, 2013.

[30] A. Dhanapal and P. Nithyanandam. An openstack based cloud testbed framework for evaluating http flooding attacks. *Wireless Networks*, pages 1–11, 2019.

[31] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra. Madam: a multi-level anomaly detector for android malware. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 240–253. Springer, 2012.

[32] R. Doshi, N. Apthorpe, and N. Feamster. Machine learning ddos detection for consumer internet of things devices. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 29–35. IEEE, 2018.

[33] C. Douligeris and A. Mitrokotsa. Ddos attacks and defense mechanisms: classification and state-of-the-art. *Computer networks*, 44(5):643–666, 2004.

[34] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–12, 2011.

[35] H. Fang, S. S. Dayapule, F. Yao, M. Doroslovački, and G. Venkataramani. Prodact: Prefetch-obfuscator to defend against cache timing channels. *International Journal of Parallel Programming*, 47(4):571–594, 2019.

[36] A. Felfernig and R. Burke. Constraint-based recommender systems: technologies and research issues. In *Proceedings of the 10th international conference on Electronic commerce*, pages 1–10, 2008.

[37] B. Feng, H. Zhou, H. Zhang, J. Jiang, and S. Yu. A popularity-based cache consistency mechanism for information-centric networking. In *2016 IEEE global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[38] Y. Feng, J. Liao, D. D. Wang, M. J. Xu, and W. B. Yin. Cache utilization to efficiently manage a storage system, May 23 2017. US Patent 9,658,965.

[39] E. Friedlander and V. Aggarwal. Generalization of lru cache replacement policy with applications to video streaming. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 4(3):1–22, 2019.

[40] J. Gantz and D. Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future*, 2007(2012):1–16, 2012.

[41] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[42] H. Gomaa, G. G. Messier, C. Williamson, and R. Davies. Estimating instantaneous cache hit ratio using markov chain analysis. *IEEE/ACM transactions on Networking*, 21(5):1472–1483, 2012.

[43] S. Gopi, V. Guruswami, and S. Yekhanin. On maximally recoverable local reconstruction codes. *arXiv preprint arXiv:1710.10322*, 2017.

[44] S. Gupta, P. Kumar, and A. Abraham. A profile based network intrusion detection and prevention system for securing cloud environment. *International Journal of Distributed Sensor Networks*, 9(3):364575, 2013.

[45] S. Han and J. Xing. Ensuring data storage security through a novel third party auditor scheme in cloud computing. In *2011 IEEE International Conference on Cloud Computing and Intelligence Systems*, pages 264–268. IEEE, 2011.

[46] H. Hu, Y. Wen, T.-S. Chua, and X. Li. Toward scalable systems for big data analytics: A technology tutorial. *IEEE access*, 2:652–687, 2014.

[47] J. Hu, J. Liang, Y. Kuang, and V. Honavar. A user similarity-based top-n recommendation approach for mobile in-application advertising. *Expert Systems with Applications*, 111:51–60, 2018.

[48] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 15–26, 2012.

[49] J. Huang, X. Liang, X. Qin, Q. Cao, and C. Xie. Push: A pipelined reconstruction i/of or erasure-coded storage clusters. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):516–526, 2015.

[50] X.-y. Huang and Y.-q. Zhong. Web cache replacement algorithm based on multi-markov chains prediction model. *Microelectron. Comput*, 31(5):123–125, 2014.

[51] Z. Huang. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery*, 2(3):283–304, 1998.

[52] A. Jain and C. Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 78–89. IEEE, 2016.

[53] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH Computer Architecture News*, 38(3):60–71, 2010.

[54] J. Ji, W. Pang, C. Zhou, X. Han, and Z. Wang. A fuzzy k-prototype clustering algorithm for mixed numeric and categorical data. *Knowledge-Based Systems*, 30:129–135, 2012.

[55] M. Jiang, C. Wang, X. Luo, M. Miu, and T. Chen. Characterizing the impacts of application layer ddos attacks. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 500–507. IEEE, 2017.

[56] S. Jin and A. Bestavros. Popularity-aware greedy dual-size web proxy caching algorithms. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 254–261. IEEE, 2000.

[57] P. Kamboj, M. C. Trivedi, V. K. Yadav, and V. K. Singh. Detection techniques of ddos attacks: A survey. In *2017 4th IEEE Uttar Pradesh Section International Conference on Electrical, Computer and Electronics (UPCON)*, pages 675–679. IEEE, 2017.

[58] M. Kantardzic. *Data mining: concepts, models, methods, and algorithms*. John Wiley & Sons, 2011.

[59] T. Karnwal, T. Sivakumar, and G. Aghila. A comber approach to protect cloud computing against xml ddos and http ddos attack. In *2012 IEEE Students' Conference on Electrical, Electronics and Computer Science*, pages 1–5. IEEE, 2012.

[60] C. C. Kaya, G. Zhang, Y. Tan, and V. S. Mookerjee. An admission-control technique for delay reduction in proxy caching. *Decision Support Systems*, 46(2):594–603, 2009.

[61] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 33–45, 2014.

[62] J. Kim, N. Shin, S. Y. Jo, and S. H. Kim. Method of intrusion detection using deep neural network. In *2017 IEEE International Conference on Big Data and Smart Computing (BigComp)*, pages 313–316. IEEE, 2017.

[63] C. Kumar. Performance evaluation for implementations of a network of proxy caches. *Decision Support Systems*, 46(2):492–500, 2009.

[64] H. Kwon, T. Kim, S. J. Yu, and H. K. Kim. Self-similarity based lightweight intrusion detection method for cloud computing. In *Asian Conference on Intelligent Information and Database Systems*, pages 353–362. Springer, 2011.

[65] W. Lee, S. J. Stolfo, and K. W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14(6):533–567, 2000.

[66] C. Li, J. Zhang, and H. Tang. Replica-aware task scheduling and load balanced cache placement for delay reduction in multi-cloud environment. *The Journal of Supercomputing*, 75(5):2805–2836, 2019.

[67] H. Li and D. Han. A novel time-aware hybrid recommendation scheme combining user feedback and collaborative filtering. *IEEE Systems Journal*, 2020.

[68] H. Li, Y. Zhang, Z. Zhang, S. Liu, D. Li, X. Liu, and Y. Peng. {PARIX}: Speculative partial writes in erasure-coded systems. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 581–587, 2017.

[69] J. Li, M. N. Krohn, D. Mazieres, and D. E. Shasha. Secure untrusted data repository (sundr). In *Osdi*, volume 4, pages 9–9, 2004.

[70] J. Li, B. Li, and B. Li. Mist: Efficient dissemination of erasure-coded data in data centers. *IEEE Transactions on Emerging Topics in Computing*, 2018.

[71] J. Li, R. J. Stones, G. Wang, X. Liu, Z. Li, and M. Xu. Hard drive failure prediction using decision trees. *Reliability Engineering & System Safety*, 164:55–65, 2017.

[72] P. Li, S. Gong, S. Gao, Y. Hu, Z. Pan, and X. You. Delay-constrained sleeping mechanism for energy saving in cache-aided ultra-dense network. *Science China Information Sciences*, 62(8):82301, 2019.

[73] W. Li, L. Galluccio, M. Kieffer, and F. Bassi. Distributed faulty node detection in dtns. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2016.

[74] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

[75] K. Ma, B. Yang, Z. Yang, and Z. Yu. Segment access-aware dynamic semantic cache in cloud computing environment. *Journal of Parallel and Distributed Computing*, 110:42–51, 2017.

[76] X. Ma and Y. Chen. Ddos detection method based on chaos analysis of network traffic entropy. *IEEE Communications Letters*, 18(1):114–117, 2013.

[77] M. Manasse, C. Thekkath, and A. Silverberg. A reed-solomon code for disk storage, and efficient recovery computations for erasure-coded disk storage. *Microsoft Research*, 2009.

[78] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*, pages 2040–2048. IEEE, 2014.

[79] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, et al. Deep android malware detection. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 301–308, 2017.

[80] B. Meng, W. Andi, X. Jian, and Z. Fucai. Ddos attack detection system based on analysis of users' behaviors for application layer. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 1, pages 596–599. IEEE, 2017.

[81] Y. Meng, L. Zhang, D. Xu, Z. Guan, and L. Ren. A dynamic erasure code based on block code. In *Proceedings of the 2019 International Conference on Embedded Wireless Systems and Networks*, pages 379–383. Junction Publishing, 2019.

[82] S. Mittal and J. S. Vetter. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, 2015.

[83] C. N. Modi, D. R. Patel, A. Patel, and R. Muttukrishnan. Bayesian classifier and snort based network intrusion detection system in cloud computing. In *2012 Third International Conference on Computing, Communication and Networking Technologies (ICCCNT'12)*, pages 1–7. IEEE, 2012.

[84] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[85] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's warm {BLOB} storage system. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 383–398, 2014.

[86] M. M. Najafabadi, T. M. Khoshgoftaar, C. Calvert, and C. Kemp. User behavior anomaly detection for application layer ddos attacks. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 154–161. IEEE, 2017.

[87] Q. N. Nguyen, J. Liu, Z. Pan, I. Benkacem, T. Tsuda, T. Taleb, S. Shimamoto, and T. Sato. Ppcs: a progressive popularity-aware caching scheme for edge-based cache redundancy avoidance in information-centric networks. *Sensors*, 19(3):694, 2019.

[88] E. Ozfatura and D. Gündüz. Mobility and popularity-aware coded small-cell caching. *IEEE Communications Letters*, 22(2):288–291, 2017.

[89] D. Patterson, G. Gibson, and R. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

[90] M. J. Pazzani and D. Billsus. Content-based recommendation systems. In *The adaptive web*, pages 325–341. Springer, 2007.

[91] D. Pertin, S. David, P. Evenou, B. Parrein, and N. Normand. Distributed file system based on erasure coding for i/o intensive applications. In *4th International Conference on Cloud Computing and Service Science (CLOSER)*, volume 1, pages 451–456. SciTePress, 2014.

[92] J. Plank et al. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software Practice and Experience*, 27(9):995–1012, 1997.

[93] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proccedings of the 7th conference on File and storage technologies*, pages 253–265. USENIX Association, 2009.

[94] R. Poddar, S. Wang, J. Lu, and R. A. Popa. Practical volume-based attacks on encrypted databases. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 354–369. IEEE, 2020.

[95] S. Podlipnig and L. Böszörmenyi. A survey of web cache replacement strategies. *ACM Computing Surveys (CSUR)*, 35(4):374–398, 2003.

[96] A. Praseed and P. S. Thilagam. Ddos attacks at the application layer: Challenges and research perspectives for safeguarding web applications. *IEEE Communications Surveys & Tutorials*, 21(1):661–685, 2018.

[97] F. L. Quilumba, W.-J. Lee, H. Huang, D. Y. Wang, and R. L. Szabados. Using smart meter data to improve the accuracy of intraday load forecasting considering customer behavior similarities. *IEEE Transactions on Smart Grid*, 6(2):911–918, 2015.

[98] M. A. Rahaman, A. Schaad, and M. Rits. Towards secure soap message exchange in a soa. In *Proceedings of the 3rd ACM workshop on Secure web services*, pages 77–84, 2006.

[99] S. Ranjan, R. Swaminathan, M. Uysal, and E. W. Knightly. Ddos-resilient scheduling to counter application layer attacks under imperfect detection. In *INFOCOM*, pages 1–14. Citeseer, 2006.

[100] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[101] F. Rezaeimehr, P. Moradi, S. Ahmadian, N. N. Qader, and M. Jalili. Tcars: Time- and community-aware recommendation system. *Future Generation Computer Systems*, 78:419–429, 2018.

[102] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238, 1999.

[103] S. Sahoo. Faulty node detection in wireless sensor networks using cluster, 2013.

[104] B. M. Sarwar, G. Karypis, J. A. Konstan, J. Riedl, et al. Item-based collaborative filtering recommendation algorithms. *Www*, 1:285–295, 2001.

[105] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 11–20. IEEE, 2015.

[106] S. Seok and H. Kim. Visualized malware classification based-on convolutional neural network. *Journal of the Korea Institute of Information Security & Cryptology*, 26(1):197–208, 2016.

[107] K. Shah, A. Mitra, and D. Matani. An o (1) algorithm for implementing the lfu cache eviction scheme. *no*, 1:1–8, 2010.

[108] J.-P. Sheu and Y.-C. Chuo. Wildcard rules caching and cache replacement algorithms in software-defined networking. *IEEE Transactions on Network and Service Management*, 13(1):19–29, 2016.

[109] K. Shvachko, H. Kuang, S. Radia, R. Chansler, et al. The hadoop distributed file system. In *MSST*, volume 10, pages 1–10, 2010.

[110] K. J. Singh and T. De. Mlp-ga based algorithm to detect application layer ddos attack. *Journal of information security and applications*, 36:145–153, 2017.

[111] I. Sreeram and V. P. K. Vuppala. Http flood attack detection in application layer using machine learning metrics and bio inspired bat algorithm. *Applied computing and informatics*, 15(1):59–66, 2019.

[112] L. Stein. The world wide web security faq, version 3.1. 2. *http://www. w3. org/Security/Faq/*, 2002.

[113] M. W. Storer, K. M. Greenan, E. L. Miller, and K. Voruganti. Pergamum: Replacing tape with energy efficient, reliable, disk-based archival storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, page 1. USENIX Association, 2008.

[114] V. L. Thing, M. Sloman, and N. Dulay. Adaptive response system for distributed denial-of-service attacks. In *2009 IFIP/IEEE International Symposium on Integrated Network Management*, pages 809–814. IEEE, 2009.

[115] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1013–1020. ACM, 2010.

[116] K. B. Virupakshar, M. Asundi, K. Channal, P. Shettar, S. Patil, and D. Narayan. Distributed denial of service (ddos) attacks detection system for openstack-based private cloud. *Procedia Computer Science*, 167:2297–2307, 2020.

[117] B. Wang, Y. Zheng, W. Lou, and Y. T. Hou. Ddos attack protection in the era of cloud computing and software-defined networking. *Computer Networks*, 81:308–319, 2015.

[118] Y. Wang, W. Chen, J. Yang, and T. Li. Towards memory-efficient allocation of cnns on processing-in-memory architecture. *IEEE Transactions on Parallel and Distributed Systems*, 29(6):1428–1441, 2018.

[119] M. R. Watson, A. K. Marnerides, A. Mauthe, D. Hutchison, et al. Malware detection in cloud computing infrastructures. *IEEE Transactions on Dependable and Secure Computing*, 13(2):192–205, 2015.

[120] H. Weatherspoon and J. D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. *Peer-to-Peer Systems*, 2002.

[121] Z. Wilcox-O'Hearn and B. Warner. Tahoe: the least-authority filesystem. In *Proceedings of the 4th ACM international workshop on Storage security and survivability*, pages 21–26, 2008.

[122] X. Wu, H. Xu, X. Zhu, and W. Li. Web cache replacement strategy based on reference degree. In *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, pages 209–212. IEEE, 2015.

[123] Y. Xiang, T. Lan, V. Aggarwal, and Y.-F. R. Chen. Joint latency and cost optimization for erasure-coded data center storage. *IEEE/ACM Transactions on Networking*, 24(4):2443–2457, 2015.

[124] F. Xie, L. Du, Y. Bai, and L. Chen. Popularity aware scheduling for network coding based content distribution in ad hoc networks. In *2007 IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications*, pages 1–5. IEEE, 2007.

[125] X. Xie, C. Wu, J. Gu, H. Qiu, J. Li, M. Guo, X. He, Y. Dong, and Y. Zhao. Az-code: An efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 230–243. IEEE, 2019.

[126] Y. Xie and S.-Z. Yu. Monitoring the application-layer ddos attacks for popular websites. *IEEE/ACM Transactions on networking*, 17(1):15–25, 2008.

[127] J. Xu, Q. Hu, W.-C. Lee, and D. L. Lee. Performance evaluation of an optimal cache replacement policy for wireless data dissemination. *IEEE Transactions on knowledge and Data Engineering*, 16(1):125–139, 2004.

[128] S. Xu, G. Yang, Y. Mu, and X. Liu. A secure iot cloud storage system with fine-grained access control and decryption key exposure resistance. *Future Generation Computer Systems*, 97:284–294, 2019.

[129] C. Ye, K. Zheng, and C. She. Application layer ddos detection using clustering analysis. In *Proceedings of 2012 2nd International Conference on Computer Science and Network Technology*, pages 1038–1041. IEEE, 2012.

[130] W. Yen and M.-F. Lee. Defending application ddos with constraint random request attacks. In *2005 Asia-Pacific Conference on Communications*, pages 620–624. IEEE, 2005.

[131] F. Yihunie, E. Abdelfattah, and A. Odeh. Analysis of ping of death dos and ddos attacks. In *2018 IEEE Long Island Systems, Applications and Technology Conference (LISAT)*, pages 1–4. IEEE, 2018.

[132] S. Zhang, L. Wang, H. Luo, X. Ma, and S. Zhou. Age of information and delay tradeoff with freshness-aware mobile edge cache update. In *2019 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6. IEEE, 2019.

[133] Z.-D. Zhao and M.-S. Shang. User-based collaborative-filtering recommendation algorithms on hadoop. In *2010 Third International Conference on Knowledge Discovery and Data Mining*, pages 478–481. IEEE, 2010.