2022 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)

Whole-File Chunk-Based Deduplication Using Reinforcement Learning for Cloud Storage

Xincheng Yuan
Department of Computer Science
San José State University
San José, CA, USA
xincheng.yuan@sjsu.edu

Melody Moh
Department of Computer Science
San José State University
San José, CA, USA
melody.moh@sjsu.edu

Teng-Sheng Moh
Department of Computer Science
San José State University
San José, CA, USA
teng.moh@sjsu.edu

Abstract - Deduplication is the process of removing replicated data content from storage facilities like online databases, cloud datastore, local file systems, etc. It is commonly performed as part of data preprocessing to eliminate redundant data that requires extra storage spaces and computing power and is crucial for data storage management in cloud computing. Deduplication is essential for file backup systems since duplicated files will presumably consume more storage space, especially with a short backup period such as daily. A common technique in this field involves splitting files into chunks whose hashes can be compared using data structures or techniques like clustering. This paper explores the possibility of performing such file chunk deduplication leveraging an innovative reinforcement learning approach to achieve a high deduplication ratio. The proposed system is named SegDup, which achieves 13% higher deduplication ratio than Extreme Binning, a state-of-the art deduplication algorithm.

Keywords: Reinforcement Learning, Deduplication, Deep Q-Network, Bloom Filter, Cloud Storage.

I. Introduction

We live in a world with fast developing technologies, most of which rely on tremendous amounts of data. According to Statista [7], a global survey company, the total amount of data created in the world grew exponentially from 2 zettabytes in 2010 to 64.2 zettabytes in 2020 and is expected to reach 181 zettabytes in 5 years, as shown in Figure 1. Demand for data has become even higher since the beginning of the COVID-19 pandemic because of the increasing number of people frequently working and entertaining at home. Therefore, the installed base of storage capacity is forecast to grow at a compound annual growth rate of 19.2% from 2020 - 2025. Therefore, plenty of systems like Google Drive, Dropbox and OneDrive were released for cloud storage supporting both manual backup and automatic synchronization of files stored locally on the OS.

Sun, Zhen, et al. [8] discovered that, for periodical file backup, the shorter the backup period is, the more likely duplicate files will be stored. Hence deduplication is critical for file backup systems to reduce cost by saving storage spaces. As one of the most adopted approaches, file chunk deduplication systems start by breaking incoming files into chunks that will be hashed for identification using functions such as SHA-1, SHA-256, MD5, etc. By comparing values obtained from hashing, which can be called "chunk fingerprints", deduplication systems would be able to identify duplicate chunks, which will be replaced by pointers to the unique copies stored. Deduplication performance can be evaluated directly using deduplication ratio (DR), defined as in Equation 1 below:

$$\mathrm{DR} = \frac{Storage\ Space\ Used\ Before\ Deduplication}{Actual\ Storage\ Space\ Used\ After\ Deduplication} \quad (1)$$

Since storage space used before deduplication never changes, the larger the deduplication ratio, the more effective the duplication was to reduce final storage space consumption.

We propose using SegDedup, a solution to effective chunk-based file system deduplication using reinforcement learning. The rest of the paper is arranged as follows. We will first show related works. Then we will demonstrate our implementation. Next, we will present the setup, results and evaluations. Finally, we conclude the paper with a description of the future works.

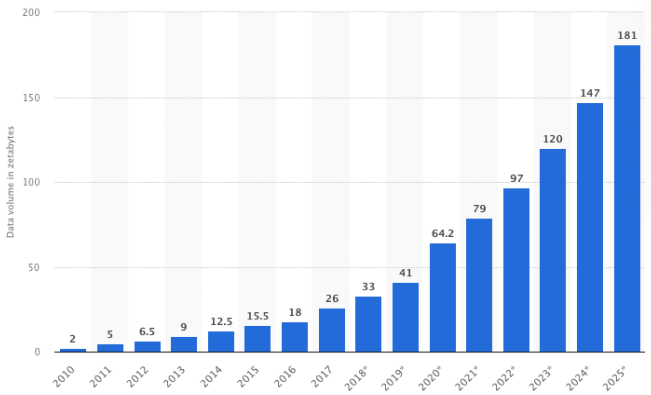


Fig. 1 Past and Estimated Future Volume of data/information Creation Worldwide from 2010 to 2025 (in zettabytes) Collected by Statista [7]

II. Background and Related Works

In this section, we provide some related background for our project. This section contains six subsections. In the first subsection, we introduce the reinforcement learning model as a special branch of machine learning mechanism while describing its connection to our project. In the second and third subsections we demonstrate Q-learning, a commonly used reinforcement learning algorithm, as well as Deep Q Network (DQN) that integrates Q-learning into deep learning. In the next subsection, we explain the challenges behind deduplication systems. In the last two subsections, we explore some existing solutions to the related problem with an emphasis on Extreme Binning, a baseline algorithm.

A. Reinforcement Learning

In our project, deduplication is not similar to a classification or progression task. No data label is available beforehand. Therefore, in this scenario, unlike supervised or unsupervised learning, we need a dynamic machine learning technique. That is why we chose reinforcement learning. Reinforcement learning is such a type of technique in which an agent, i.e., the learning machine, is placed in an interactive environment from which it learns with the help of feedback from its own actions. The agent learns from experience to improve its action over time. The goal is to find a model that maximizes the cumulative rewards of the agent. Figure 2 illustrates the framework of reinforcement learning [2].

.

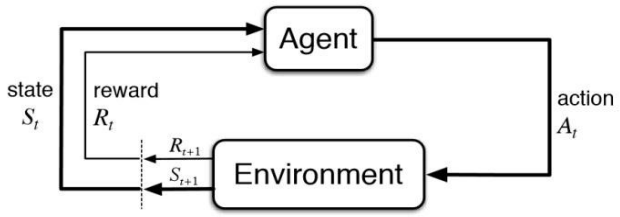


Fig.2 Reinforcement Learning Framework

B. Q- Learning

Q-learning is a branch of reinforcement learning that doesn't require a model. In this type of algorithm, we define a function **Q (s, a)** to represent the discounted future reward of an action in a given states. We name this function Q-function because it reflects the "quality" of an action in the given state. The goal of the learning process is to find the action generating maximum Q-value, so that we can obtain the corresponding policy π as:

$$\pi(s) = argmax_a\, Q(s,a) \tag{2}$$

To get the Q-function from a known transition $\{\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{r}, \boldsymbol{s'}\}$, the Bellman equation as shown in Equation 3 comes in handy. In the equation, the Q-function is expressed as a linear function of the maximum possible Q-values that can be obtained from the actions and states afterwards, where γ is the discount factor.

$$Q(s,a) = r + \gamma\, max_{a'}\, Q(s', a') \tag{3}$$

In the above equation, $\boldsymbol{Q}(\boldsymbol{s'}, \boldsymbol{a'})$ is just an estimation that could be from a very early stage of learning. As more iterations of learning are conducted, the estimation will get more accurate since it is shown that if update is performed using the above equation enough times, the Q-function will converge toward the "true Q-value" [5].

C. Deep Q-Network (DQN)

In Q-learning, a Q-Table is created to store Q-values for each action and state. The Q-Table is updated during the process as described in the previous subsection. However, in environments with many possible actions and states, a larger Q-Table is required, of which the updates and lookups become more expensive.

In 2013, Minh et al [6] introduced DQN that performs Q-learning with two neural networks instead of a Q-Table. The two neural networks referred to as Prediction Network and Target Network are similar in structure, but each performs a slightly different function. The Prediction Network takes a state S_t at time t as input and calculates the Q-Value for each action in the action space. The Target Network does the same calculation but takes as input the state S_{t+1} denoting the state of the environment after current action is performed, which is simply obtained by synchronizing with parameters of the Prediction Network. The Q-value calculated using the target network can be used as input back to the Prediction Network in place of $\boldsymbol{Q}(\boldsymbol{s'}, \boldsymbol{a'})$ in Equation 3.

Since learning with DQN is not supervised as no labels are provided to the agent beforehand, the agent needs to figure out on its own the best Q-Value under each given state. To achieve such goals, the agent takes advantage of previous experiences stored in a local memory, which is often referred to as Experience Replay. The experience replay often keeps track of rewards and actions under previous state transitions, which is then fed to the Q-network for mini-batch retraining. The Q-Network is then updated accordingly. The Experience Replay allows the agent to learn from various samples of past experiences to avoid overfitting.

In a deduplication task, file chunk fingerprints can hardly be labeled as "duplicate" or "unique" beforehand. However, a learning agent is able to judge if a chunk is unique by comparing the newly reserved chunk fingerprints to those observed previously. The learning agent could be trained to judge the chunks as if it's playing a game in which it's given the goal to select to store on to disk the fewer duplicated chunks as possible. The agent could be given straightforward rewards for its decision by examination to tell if a duplicate chunk is stored or a unique chunk is missed as the consequence. Therefore, we were inspired to adopt Q-learning techniques on our project.

D. Related Works

Data and/or file deduplication has become a popular research topic in recent years. In this subsection, we explore some of the existing algorithms proposed and implemented for the deduplication task.

In 2016, Harnik et al [4] proposed an algorithm attempting to boost the deduplication efficiency by investigating only the duplicate chunk distribution of randomly sampled data using DFH (Duplication Frequency Histogram) and estimating that of the entire dataset with a mapping. A sampling method was embedded in the algorithm with moderate memory usage, where the sample chunks were sampled further in fraction. By iterating over all chunks inside, a histogram would be maintained for the base sample and finally extrapolated to the sample as an estimated DFH. As a result, the authors claimed that the deduplication ratio would converge within generally 3% to 20% sample size at 1MB granularity for different datasets. The most significant contribution of this research is to avoid full scan in deduplication ratio estimation. The tradeoff, however, would be tuning the sample size.

In another paper published in 2019 [8], the researchers implemented a chunk deduplication program in Java starting from chunking and generating chunk Id's, and then pass chunk ids together with their metadata through a B-tree, and finally condense into memory a list of unique chunks. The authors claimed that the algorithm reduced RAM usage up to more than 90%, however for file system deduplication with much larger data size than the genomic data the authors dealt with, using such data structure may cause much extra RAM overhead for indexing. In a recent research paper [11], the authors implemented a file chunk deduplication program in context of a multi armed bandit problem, in which they used segments of chunks as the deduplication unit and used the context tables of segments as input. The agent would generate champion segments whose total reward is maximum, and whose successive segments will be prefetched into the cache. The authors claimed that their algorithm was able to remove 2% ~ 10% more redundant data.

E. Extreme Binning

In 2009, Bhagwat, Deepavali, et al proposed Extreme Binning, a deduplication pipeline for chunk-based file backup [1]. In their system, several backup nodes can work in parallel processing chunked files by conducting lookups on RAM before filtering out unique chunks to the disk.

As illustrated in Figure 3, in Extreme Binning deduplication system, each backup node maintains a primary index containing pointers to individual containers in the disk called bins, each keeping an index of chunk fingerprints or IDs, as well as their corresponding sizes. Once a chunked file, of which a hashed fingerprint is calculated as the individual ID for each chunk, is received by the node, all the containing chunks will be accessed to determine a mapping from the file to a representative chunk ID. In the paper, the minimum chunk fingerprint of containing chunks for every file is selected as the file's representative chunk ID, which is based on the Broder's theorem. Deduplication is performed by checking representative chunk IDs against all representative chunk IDs recorded in the primary index in RAM. At the end of the entire deduplication process, all bins containing deduplicated chunk IDs, as well as actual data chunks and file recipes are written to the disk.

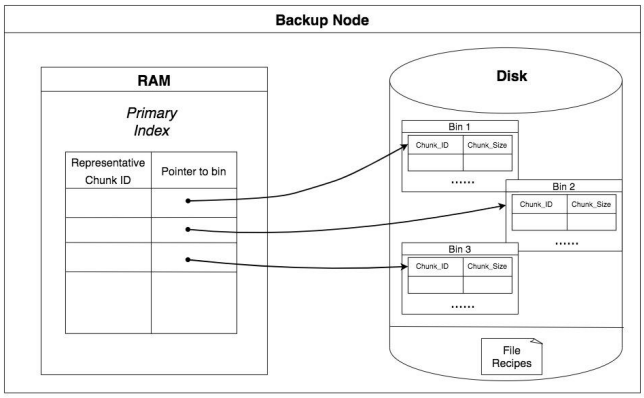


Fig.3 Extreme Binning Backup Node Architecture

Researchers claim that the Extreme Binning allows only one disk access for chunk lookup per file because it doesn't require locality. Performance showed that storage space consumed after deduplication with Extreme Binning was considerably less, yet there was still quite a gap between the deduplication ratio achieved and that of perfect deduplication, in which case each chunk fingerprint was compared to every other one such that no duplicate chunks are stored.

We successfully reproduced and tested the Extreme Binning algorithm in Python based on the original paper, which we will use as a baseline.

III. Design and Implementation

A. System Information

This project was implemented and tested on a MacBook Pro with 4 cores Intel ® Core i7 CPU at 2.5 GHz running macOS High Sierra version 10.13.6. The machine has 16 GB of installed 1600 MHz DDR3 memory and 500 GB Macintosh hard drive for flash storage.

B System Architecture

From input files retrieval to complete deduplication, we implemented three major modules for the pipeline, namely the Data Extraction, Data Preprocessing and Main module. Figure 4 demonstrates the module setup and data flow.

Data Extraction: Once fetched from the dataset, the source file containing hashed fingerprints of files and belonging chunks of a snapshot were accepted by the feature extractor of the data extraction module as input to a command line tool named hf-stat provided on FSL website to parse the source files. Feature extractor executed hf-stat to obtain an output of chunk fingerprints together with other metadata including file path, size, access, creation and modification time, permission bits, user ID, group ID, device ID and Inode Number of each file in the current snapshot. The feature extractor filters out the chunk fingerprints of a single file, one of the most critical attributes for chunk identification, at once from the hf-stat output. In the

meantime, the feature extractor grabed the size info of each for deduplication analysis, and whole file fingerprints if required by any algorithms utilizing the module. The data returned by the feature extractor that was passed forward simulated the generation of a stream of file chunk data. Once the module was invoked, chunk data of a new file was returned until the last file of the last snapshot was accessed.

Data Preprocessing: The data preprocessing module was responsible for chunk segmentation. Like many other existing deduplication algorithms, SegDedup practiced deduplication on chunks grouped into segments to improve the efficiency in consideration of a significantly enormous number of chunks in just one single snapshot. However, segmentation could reduce deduplication sensitivity unless duplicate chunks are located exactly in duplicate segments, otherwise detection of identical chunks from different segments would fail without segment-

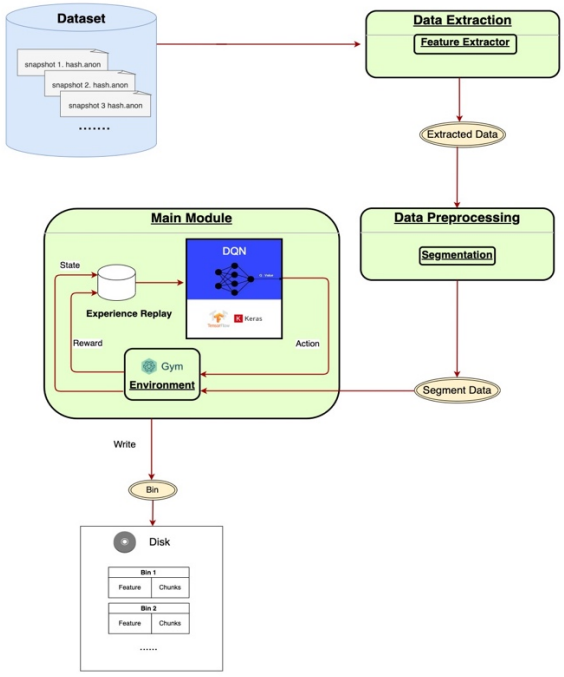


Fig.4 SegDedup System Architecture

wise comparison. To tackle the above problem, SegDedup first has already ensured that all chunks making up a segment are from the same file by restricting the input of segmentation function to the output chunk data of only one file once from the data extraction module. It's guaranteed that no segment contains chunks of more than one file. Second, each segment is mapped to a feature resembling the representative chunk ID in Extreme Binning algorithm to enable capture of potential segment similarity. The task of the data preprocessing module, however, is just to segment the chunks of a file according to a defined size threshold. If the combined chunk size of the current segment reaches or surpasses the threshold, a new segment will be initiated to accommodate excess chunks. As a result, when its task is done, the data preprocessing module will yield a list of segments belonging to the input file where the total chunk size of each segment will never exceed the threshold. To determine the feature of an individual segment we consulted the method used to fix the representative chunk IDs in Extreme Binning. Nevertheless, for added confidence in assessment of segment similarity, instead of using only the minimum chunk fingerprint, we introduced both the maximum chunk fingerprint together and the total size of containing chunks into the segment.

Size	min_ chunk_ fingerprint	max_ chunk_ fingerprint	chunk 1 _hash	chunk 2 _hash	...	chunk n _hash

Fig.5 Features Stored In a Segment of Chunks

Main module: The main module performs the core functions of deduplication on segments obtained from the data preprocessing module. The module serves as the core of SegDedup where DQN construction and reinforcement learning will happen. Performance of reinforcement learning depends mainly on the set up of the environment in which the agent could find guidance on what actions can be taken, how to take the actions, and what will happen after each action is taken. Furthermore, the agent will need to interact with the environment by taking actions to receive feedback of rewards. In the rest of the subsection, we will show the implementation of the main module in greater detail.

C. Implementation

1) DON and Interactive Environment Setup: To perform deduplication with reinforcement learning, we constructed DQN in the main module. The DQN whose purpose is to evaluate respective Q-values of each individual permitted action of the agent under input states has a three-layer structure: a ReLU layer for input, a hidden layer, and a linear output layer. The prediction of q-value and retraining performed on DQN significantly relies on the experience replay as source of input data. To be specific, the experience reply keeps five values, the current state of the environment, the current action of the agent, the agent's reward earned by taking the action, the state of environment after current action is taken, and the done flag to tell if the environment sends a termination signal to stop the agent from proceeding with any more interactions.

During interaction, SegDedup keeps two pieces of memory: cache, and a Bloom filter [3] for segment comparison and lookup. The cache was designed with a size threshold to temporarily hold information including the feature, size in bytes, score of hits, and containing chunks of some segments throughout the entire deduplication process. For example, a valid entry in the cache could look like:

0xAA389A7D3FE1,12288, 2, [0x3430019ba186, 0x4ffa1ca6f0b7, 0x44d5c81c10d9]

Reaching the maximum cache size capacity will trigger an eviction process in which certain segments will be expelled from the cache to the Bloom filter. Claim that an element belongs to a set using the Bloom filter could lead to false

positive error, the probability of which can be calculated with the following equation:

$$Pr\ (false\ positive)\ \approx (1 - e^{-k\frac{n}{m}})^k \quad (4)$$

Nevertheless, if at least one of the bits at an index among the **k** hash function values of an item is set to 0, it means that the item is definitely not present in the set. In other words, a verdict on the non-existence of an element in a set according to the Bloom filter will never result in false negative error. Fortunately, given **n**, the number of elements to be inserted, and **ε**, the desired false positive probability, we could deduce the optimal length of Bloom Filter **m** using Equation 5:

$$m = -\frac{nln\ \varepsilon}{(ln2)^2} \quad (5)$$

The number of hash functions is then:

$$k = \frac{m}{n} ln\ 2 \quad (6)$$

The Bloom filter used in this project supports automatic detection of the optimal settings **m** and **k** that limits the false positive probability as small as 0.01 taking advantage of the above equations. The Bloom filter is necessary for the purpose of keeping a record of evicted segments because of cache getting overloaded for comparison and feedback during agent interaction with the environment.

2) Reinforcement Learning: In this subsection, we give detailed description how the agent interacts with the environment and how it's trained using DQN, which is the core of the entire deduplication process.

For each step of interaction, a new file would be fetched by the data extraction module and segmented by the data preprocessing module. The segments would then be available to the agent as a new observation in the observation space of the environment before the agent is ready to take an action. Next, on the observed segments, the agent is given two possible options of actions which we define as the following:

a_0: skip the segments for now
a_1: select the segments for cache storage

There are two possible outcomes for each of the possible actions. When the agent chooses to take action **a_1**, it could either have selected a segment already present in cache, or a new segment that didn't appear before. The former case is less favorable because a duplicate chunk was selected. In case that action **a_0** is taken, the segment skipped could be a duplicated segment, or it could have skipped a unique segment. We don't encourage the latter case even though the skipped segment could be encountered again later, because the probability is not 100%. In summary, the objective is to train the agent to try to keep selecting the unique segments while neglecting duplicated segments more frequently and accurately throughout the process.

Inside the environment class, we give specific definitions of both actions. For each action the agent selects, the segments would mainly be checked against the Bloom filter followed by the cache. Figure 6 illustrates the entire process of agent interaction with the environment. We provide detailed explanation to all six phases involved in one step of interaction as the following:

Phase1: Feature Lookup As mentioned previously, each segment was mapped to a feature using its minimum and maximum chunk fingerprint. During reinforcement learning, we use the feature again to represent the segment so that instead of storing entire segments to the data structures, we could only store the feature entity. For further simplicity, we use the Szudzik pairing function [9] to encode the two feature values into a single value. The function, shown as Equation 7, is proven to be effective in uniquely encode two integers while keeping the encoded result relatively small for storage friendliness. Once the encoded feature is calculated, it will be checked against the bloom filter.

$$Pair\ (x, y) = x^2 + x + y \quad (7)$$

Phase 2: Reward or Check Cache If the search of an encoded feature in the Bloom filter returns True, there's a high probability that it was in the Bloom filter before. If the feature was in Bloom filter, it means that the feature was probably evicted to the Bloom filter at some point before, which indicates that the segment might have existed before in cache, making the segment suspicious for duplicate. The agent is then rewarded or punished according to the chosen action since skipping a duplicated segment is more desired than selecting it. The current turn of the agent will end here because checking the segment against the cache is not necessary in this case. Otherwise, the segment will be checked against the cache next step.

Phase 3 and 4: Load to Cache If the Bloom filter search returns False, then the segment was not evicted before. Hence the segment will be looked up in the cache for further investigation. If the segment feature is in the cache, the hit score of the feature will be incremented by one to be used for reward. Simultaneously, chunks of the segments will be appended to the associating chunks of the feature while the total chunk size gets updated accordingly. On the contrary, if the segment cannot be found in cache, we can then confirm that the segment is unfamiliar to the cache. In consequence, a new entry for the current feature will be inserted to the cache. The same insertion step occurs in case the action is to skip the current segment to avoid skipping of a unique segment, that is, we will correct the agent's mistake in such cases to ensure that in the end all segments were given a chance to be processed at some point in duration of entire deduplication task. The reason is that we consider that incompleteness caused by omission of a chunk is a worse outcome than introduction of a duplicated chunk.

Phase 5: Reward After step 3 and 4, the environment will finally give feedback to the agent. For an existing segment in the cache, the reward will correspond to its hit score, which will be multiplied by a factor, namely 1 or -1, depending on the action as the preferable action would be to skip it. If a new entry

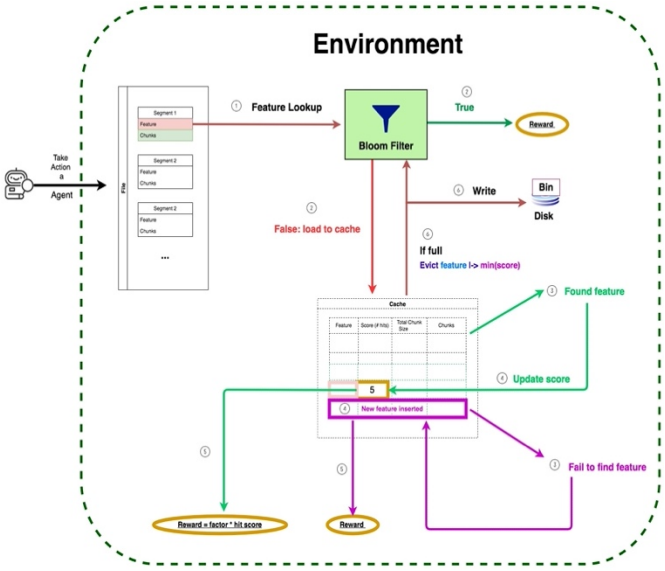


Fig.6 Agent-Environment Interaction Process

was inserted to cache as a result, the agent will be given a constant reward, simply for discovering a new segment.

Phase 6: Eviction The entire interaction is done after step 5. When the current action, state and reward info are computed and added to the experience replay, the environment would be waiting for the agent's next action. However, if the cache becomes full after step 5, the environment will trigger an extra eviction process targeting the feature with lowest hit score. The eviction will start from recording the feature in the Bloom filter. Then the unique chunks filtered out from those associated with the feature would be gathered to a bin to be written to the disk.

Before the environment sends a done signal, after each complete step of interaction, the state of the environment needs to be logged to the experience replay. To represent the state, we pick the first segments passed into the environment as a representative, whose minimum chunk fingerprint, maximum chunk fingerprint and size as mentioned in section 3.4.1 are reserved. All other four values would be inserted to the experience replay as well. If the experience replay has adequate number of recordings for a pre-defined batch size, it will be sampled to get a batch. For each recording in the batch, the Prediction Network would be used to predict the q values for each possible action given the current state. In the meantime, predicted q values for each possible action given the next state would be obtained from the Target Network. Then the target q value $\mathbf{t}_q$ could be fixed via Equation 3 using the maximum q value from those predicted for each action by the target network. Suppose that the prediction result of the Prediction Network and Target Network are kept in vectors named $\mathbf{V}_p$ and $\mathbf{V}_t$ respectively, then in $\mathbf{V}_p$ the q value of the action from the current recording would be replaced by $\mathbf{t}_q$ such that the current state and updated $\mathbf{V}_p$ could be used as predictor and target variables to retrain the Prediction Network for one epoch.

IV. Experiment and Results

A. Dataset

To test the deduplication performance, we adopted two datasets for the project, namely FSLhomes and MacOS. Both datasets are from File systems and Storage Lab (FSL) website [12]. FSL comprises researchers and students from the computer science department of Stony Brook University focusing their research in operating systems on file systems, storage, security, and networking. According to the website, FSLhomes contains a snapshot of students' home directories from a shared network file system collected on an approximately daily basis from late 2011 to early 2014. The files consist of source code, office documents, virtual machine images, etc. MacOS contains snapshots collected from 2011 to 2016 on a Mac OS X Snow Leopard server running services including SMTP:Postfix, MySQL for Bugzilla, Calendar server (CalDAV), Wiki server, etc. Files in snapshots from both datasets were collected with chunks in average sizes ranging from 2KiB to 128KiB, which were hashed using MD5. Because of time and system storage space restriction, we could only select a small subset of the datasets for experiment and analysis purposes. For FSLhomes, we collected various random subsets of all 127 snapshots of the user numbered 005 in 2012 and renamed them according to subset cardinality. We filtered out only snapshots with only 2KiB average chunk size considering higher ratio of duplicate chunks under smaller chunk granularity. As shown in Table 1, we collected in total three subsets, which are Homes_User005_04 containing four snapshots selected at random, Homes_User005_06 containing six snapshots and Homes_User005_08 containing eight snapshots spanning all snapshots comprising snapshots of version number 2^i along with the last snapshot, namely snapshots 1, 2, 4, 8, 16, 32, 64, 127. Notice that the total chunk size of Homes_User005_06 is bigger than that of Homes_User005_08 even though it has fewer snapshots because half of the snapshots from Homes_User005_06 are from version 100+, which were collected at the very end of the year making their individual sizes larger for the accumulated data throughout the year carried with them.

For MacOS, we focused on a snapshot from November 2011 that was collected with 128KiB average chunk size. Despite that only one snapshot was adopted, both the overall chunk count and file size are remarkably greater than that of an average snapshot from FSLhomes since stored on the same Mac server were files generated by miscellaneous users, which provided us with a scenario of cross-user file deduplication for experiment.

TABLE I
Dataset Summary

Dataset	Extreme Binning	SegDedup	increment(%)
Homes_User005_04	1.662	1.881	**13.177**
Homes_User005_06	2.328	2.593	11.383
Homes_User005_08	3.285	3.641	10.837
MacOS	1.135	1.125	-0.881

Table 1 provides a summary of total chunk count, total chunk

sizes, as well as the perfect deduplication ratio of all four datasets. Notice that Homes_User005_06 has fewer snapshots but more chunks that makes larger total chunk size than Homes_User005_08 because half of the snapshots in Homes_User005_06 collected on the last quarter of the year contains accumulated information stored on the server year-round.

B. Experiment Setup

Starting from the first snapshot of the dataset, the segments were continuously passed to the environment as a data stream. The agent's task was simply to select a proper action to perform on the incoming segments of each file. Our expectation is that the agent will get more familiar with different segments encountered overtime such that it is increasingly adept at keeping unique segments while skipping duplicate ones.

For experiment, we kept the segment size threshold at 16 KB, which means each segment could hold on average four chunks. We also set the size threshold of cache to 2 MB. For reinforcement learning, we initiated the exploration rate with a top value 1, which then slowly decayed at a rate of 0. 9995. As a result, the agent takes action mostly randomly at the early phase but gradually explores less, instead it more frequently relies on Q-values learned to determine the better action. In addition, we configure the DQN to be retrained using stored actions and states from experience replay in batches of 32. The discount factor was set to 0.99, which is large because clearly in deduplication task all segments selected or skipped in the future significantly depend on what segments had been selected in the past. In this case it's necessary to consider the future reward to a great extent for determination of current Q-value.

The agent continued to take actions as long as segments were passed into the environment from the data preprocessing module. When the data extraction module sent a termination signal indicating all snapshots have been processed, it marked the completion of the entire deduplication operation.

C. Deduplication Ratio

After running SegDedup on each of the four datasets: MacOS, Homes_User005_04, Homes_User005_08 and Homes_User005_06 respectively, we obtained the deduplication ratios calculated using Equation 1 from the introduction section. In comparison, we ran the deduplication task using the same datasets with Extreme Binning to test the quality of deduplication performance. We also collected perfect deduplication ratios of all datasets using the brute-force chunk to chunk comparison approach. To compute deduplication ratio of SegDedup, we used the data extraction module to add up sizes of chunks from each file to get the total size of original chunks; we counted the size of chunks in the segment written to disk every time eviction happened during agent interaction. Dividing the total size of original chunks by the total size of chunks written to disk, we obtained results as shown in Table 2.

TABLE 2
Deduplication Ratio

Dataset	Total Chunk Count	Total Chunk Size (GB)	Unique Chunk Size (GB)
Homes_User005_04	1974746	4550.0	2321.4
Homes_User005_06	4705262	10413.0	2720.2
Homes_User005_08	3148297	6956.4	2559.4
MacOS	79273	3676.4	3105.1

According to the results, SegDedup outperforms Extreme Binning on all datasets except the MacOS. In general, SegDedup was able to achieve a deduplication ratio around 10% higher than Extreme Binning.

On Homes_User005_04 SegDedup made a highest improvement of 13.1% on deduplication ratio compared to Extreme Binning. In the case of MacOS, SegDedup obtained a deduplication ratio 0.881% slightly lower than Extreme Binning, which is way less than the percentage difference of Extreme Binning and SegDedup deduplication ratio in all other cases. The lower deduplication ratio on MacOS is possibly related to the fact that MacOS has the lowest perfect deduplication ratio because files in its single snapshot were created by a diversity of users. In such a scenario the agent would encounter unique segments more frequently than the duplicate ones, which might give the agent a harder time to get sensitive enough to the duplicate segments so that it could avoid more often the mistake of selecting duplicate segments.

Figure 7 reflects the deduplication performance in terms of storage size differences when conducting no deduplication, deduplication using Extreme Binning, SegDedup and perfect deduplication. Figure 8a once again shows SegDedup's ability to reduce the storage spaces required further than Extreme Binning in most cases. On MacOS, Extreme Binning performs only slightly better. The best performance was on Homes_User005_06 where SegDedup saved 7553 MB of storage space from the original. According to Figure 8b, the trend of size reduction by SegDedup is closer to that of perfect deduplication than Extreme Binning.

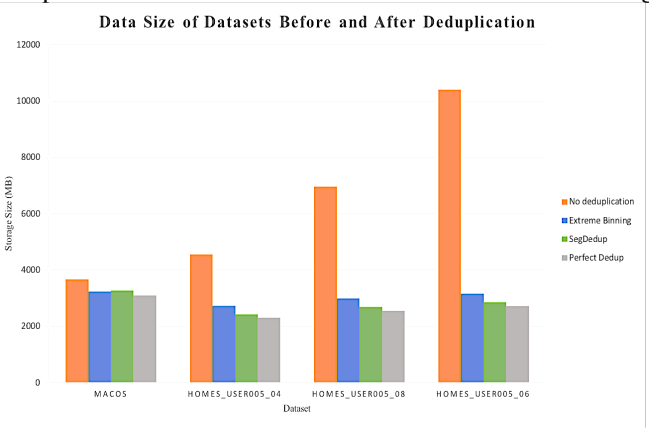
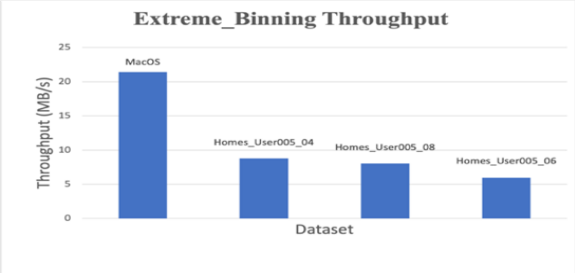


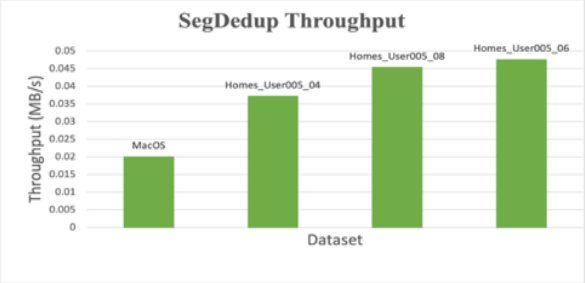
Fig. 7 Deduplication Performance by Size Difference

D. Throughput

Figure 8 illustrates throughput measurement by MB/s. Despite the fact that SegDedup takes a longer time to perform reinforcement learning, when datasets are arranged by sizes in ascending orders as shown in Figure 8a, the throughput result of Extreme Binning manifests a trend of decline while that of SegDedup remains approximately constant. Therefore, increasing throughput would result in universal acceleration in processing speed on any data size.



(a)



(b)

Fig. 8 Throughput

V. Conclusion and Future Work

We designed and implemented SegDedup, a whole-file chunk deduplication system. By integrating into the system DQN with the Gym environment framework, we explored the possibility of applying pure reinforcement learning on such an onerous and complex task. Using subsets of snapshots extracted from two real-world datasets, we achieved a maximum of 13% increase in deduplication ratio compared to Extreme Binning, an existing state-of-the art algorithm.

Experiment results suggested that our system is sufficient for small-scale deduplication. For deduplication on a greater volume of data, further investigation is needed in the design of learning and interaction process to improve the throughput. More work is also needed to minimize the tradeoff between deduplication ratio and other factors including throughput to make SegDedup applicable in a wider scope.

References

[1] D. Bhagwat, K. Eshghi, D. D. E. Long and M. Lillibridge, "Extreme Binning: Scalable, parallel deduplication for chunk-based file backup," *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009*, pp. 1-9, doi:10:1109/MASCOT.2009.5366623.

[2] S. Bhatt, "5 Things You Need to Know about Reinforcement Learning.", *Kdnuggets*, 2018. [Online]. Available: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html

[3] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey". in *Internet mathematics 1.4*, pp. 485-509, 2004.

[4] D. Harnik, E. Khaitzin, and D. Sotnikov, "Estimating unseen deduplication— from theory to practice." in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*, pp. 277-289, 2016.

[5] T. Matiisen, "Demystifying Deep Reinforcement Learning", *Computational Neuroscience Lab*, 2015. [Online]. Available: https://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller, "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602*, 2013.

[7] "Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025", *Statista*, 2021. [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/

[8] Z. Sun, G, Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, E. Zadok, "Cluster and single-node analysis of long-term deduplication patterns.", *ACM Transactions on Storage (TOS)* 14.2, pp. 1-27, May 2018.

[9] M. Szudzik. "An elegant pairing function." *Wolfram Research (ed.) Special NKS 2006 Wolfram Science Conference*, pp. 1-12, 2006.

[10] T.T. Thwel, and G.R. Sinha, "Efficient Data Deduplication Mechanism for Genomic Data." *CSVTU International Journal of Biotechnology, Bioinformatics and Biomedical 4.2*, pp. 52-58, 2019

[11] G. Xu, B. Tang, H. Lu, Q. Yu and C. W. Sung, "LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication," *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, 2019, pp. 299-310, doi: 10.1109/MSST.2019.00010.

[12] FSL Traces and Snapshots Public Archive. [Online]. Available: https://tracer.filesystems.org/

[13] "Getting Started with Gym", *Gym*, 2021. [Online]. Available: https://gym.openai.com/docs/

[14] OpenAI: https://openai.com/

[15] "TensorFlow Core Documentation for Python", *Tensorflow*. [Online]. Available: https://www.tensorflow.org/versions/r2.4/api_docs/python/t