LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication

Guangping Xu*†‡, Bo Tang*†‡, Hongli Lu*†‡, Quan Yu §and Chi Wan Sung¶
*School of Computer Science and Engineering, Tianjin University of Technology, Tianjin, China
†Key Laboratory of Computer Vision and System, Ministry of Education, Tianjin, China
‡Tianjin Key Laboratory of Intelligence Computing and Novel Software Technology, Tianjin, China
Email:xugp@email.tjut.edu.cn
§School of Information Engineering, Wuhan University of Technology, Wuhan, China
Email:yuquan@whut.edu.cn
¶Department of Electronic Engineering, City University of Hong Kong, Hong Kong, China
Email:albert.sung@cityu.edu.hk

***Abstract*—In this paper, we present a learning based data deduplication algorithm, called LIPA, which uses the reinforcement learning framework to build an adaptive indexing structure. It is rather different from previous inline chunk-based deduplication methods to solve the chunk-lookup disk bottleneck problem for large-scale backup. In previous methods, a full chunk index or a sampled chunk index often is often required to identify duplicate chunks, which is a critical stage for data deduplication. The full chunk index is hard to fit in RAM and the sampled chunk index directly affects the deduplication ratio dependent on the sampling ratio. Our learning based method only requires little memory overheads to store the index but achieves the same or even better deduplication ratio than previous methods.**

In our method, after the data stream is broken into relatively large segments, one or more representative chunk fingerprints are chosen as the feature of a segment. An incoming segment may share the same feature with previous segments. Thus we use a key-value structure to record the relationship between features and segments: a feature maps to a fixed number of segments. We train the similarities of these segments to a feature represented as scores by the reinforcement learning method. For an incoming segment, our method adaptively prefetches a segment and the successive ones into cache by using multi-armed bandits model. Our experimental results show that our method significantly reduces memory overheads and achieves effective deduplication.

***Index Terms*—Deduplication; Reinforcement learning; Data prefetching; Chunk index**

I. Introduction

The amount of stored data in large-scale storage systems is growing explosively. It is desirable to have some techniques to reduce and compress redundant data. As an effective technique to data reduction, data deduplication has been received much attention and become a standard component in some modern storage products [1], [2], [3]. At present, the most widely used approach eliminates duplicate data at the chunk-level in large-scale storage systems [4], [5], [6], [16]. In these systems, an incoming data stream is broken into fixed or variable sized chunks and then each chunk is uniquely identified and duplicate-detected by a cryptographically secure hash signature (e.g., SHA-1, MD5), called fingerprint [17]. So far the data deduplication technique has at least two benefits in reducing storage space by eliminating duplicate data and data transfer latency for distributed systems [4], [21].

During the chunk-based deduplication process, a well-known challenge is how to build an efficient fingerprint indexing to help identify duplicate data chunks. It is impractical to keep such a large index in RAM and a disk-based index with one seek per incoming chunk is far too slow for large-scale storage. Such a well-known challenge is called the chunk-lookup disk bottleneck problem [17], [2]. For example, for a unique dataset of 1PB, assuming an average chunk size of 8KB, it generates about 2.5TB SHA-1fingerprints (*i.e.*, 160 bits each chunk). The fingerprint indexing structure critically affects the deduplication performance in terms of lookup latency and deduplication ratio since only part of these fingerprints can be accessed each time. Therefore our goal of this work is to illustrate how to implement an efficient fingerprint indexing structure at a low in-memory cost for deduplication systems.

Some solutions to the chunk-lookup disk bottleneck problem have been proposed in the literature. At first, Zhu et al. [17] address the disk bottleneck problem in DDFS by using an in-memory bloom filter and caching index fragments, where each fragment indexes a set of chunks found together in the input. And then, as a milestone work, sparse indexing [18] improves DDFS memory utilization by sampling the index of chunk fingerprints in memory, which reduces the memory usage to less than half of that in DDFS. Extreme Binning [21] improves deduplication scalability by exploiting the file similarity to achieve a single on disk index access per file for chunk lookup. In this approach, file similarity can be determined by comparing the IDs of a subset of chunks, allowing similar files to be grouped together in backup workloads by sampling larger pieces of a stream. Silo [19], [20] is a hybrid solution which jointly exploits similarity and locality by first exploiting similarity of a group of chunks to reduce the space of primary index in RAM and then mining locality to enhance duplicate detection through similarity detection. To explore the locality in these previous methods, it usually breaks the data stream into segments, chooses a few of the most similar segments that have been stored previously by sampling the index, and then deduplicates each segment against only its chosen few segments. Thus the lookup of a full chunk index is avoided by the sampling.

From the state-of-the-art work,**the following observations motivate this work**: *First*, the memory overheads directly depend on the sampling ratio for previous methods. In the other words, the sampling ratio (denoted as r) is proportional to the memory overheads and further affects deduplication ratio directly. At the same time, to decrease memory overhead, it needs to sample features in a segment with a high deduplication ratio [18]. Consider the following extreme case: When only one fingerprint is chosen as the feature of a segment, it results in the least memory overheads but the similarities between segments may be meaningless. *Second*, the mapping relationship of the sampled fingerprints and the segments are rather static in previous methods. It is lack of an adaptive feedback mechanism to adjust the mapping relationship to reflect the dynamics of incoming data streams. In the recent work [15], they analyze the deduplication patterns for a long-term data evolution from the users' point of view. It suggests that even similar users behave quite differently, which should be accounted for in future deduplication systems. *Third*, the similarity-based deduplication often uses Top-k (*i.e.*, selects the k most similar segments) to choose a segment to prefetch into cache among many candidates in order to check a chunk duplicate or not. Thus it greatly influences performance how to select a segment into cache. Similar to our work, HANDS [14] dynamically prefetches fingerprints from disk into cache according to working sets statistically derived from access patterns. The method relies on a domain agnostic grouping methodology which the computational cost in data training may not be overlooked. Our approach aims on the same goal but in a rather different way. We propose a simple reinforcement learning method to learn how to prefetch a segment dynamically. The proposed learning based algorithm makes a trial-and-error prefectching and then gives a delayed reward during the data stream evolution. In our work, we formalize the prefetching problem in the reinforcement learning framework and validate its effectiveness by experiments with various real-world datasets.

The main **contributions** in this paper are as follows:

(1) As the most important contribution of the paper, we propose a new deduplication framework based on reinforcement learning for data deduplication, called LIPA. To the best of our knowledge, this paper is the first learning-based deduplication, which can be implemented in a simple and effective way. The methodology is to explore locality in context of data streams; that is, an incoming segment may share the same feature with previous neighboring segments. We train the locality relationship and deduplicate each incoming segment against only a few of its previous segments. This method incurs significantly less memory cost than existing solutions while keeping the same or better deduplication performance in terms of lookup latency and deduplication ratio.

(2) We implement the proposed algorithm in a deduplication system and evaluate its performance via various real-world datasets. Compared with the state-of-the-art schemes, the experimental results demonstrate that our algorithm obtains better restore performance while keeping the desired deduplication ratio.

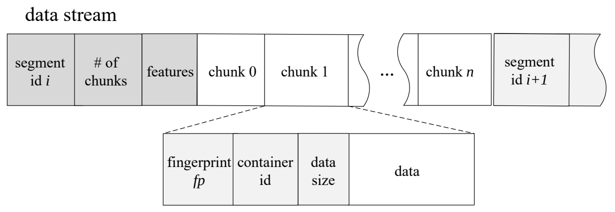

Fig. 1. Terms and their logical relationship in the segment-based deduplication system.

The rest of this paper is organized as follows: In Section II, we present some preliminaries. In Section III, we describe our proposed learning based deduplication approach. In Section IV, we show our experimental results with real data. Finally, we describe related work in Section V and our conclusions in Section VI.

II. Preliminaries

In this section, we first give some terms used in data deduplication and then introduce briefly the reinforcement learning used in our approach.

A. Segment-based Data Deduplication

In order to describe our approach, we adopt the same terms used in the survey papers [1], [2], [3]. In a segment-based deduplication system, as shown in Fig. 1, the data stream is broken into non-overlapping variable-sized *chunks* by a chunking algorithm and each chunk is identified by a hash value called *fingerprint*; and then certain chunks are aggregated into a *segment* by a segmenting algorithm, which transforms the chunk sequence into a sequence of segments. In this paper we follow the Two-Threshold Two-Divisor (TTTD) chunking method and the Content Defined Segmenting (CDS) method [18], [19]. Typically, the average size of a chunk is 4KB, the average number of chunks in a segment is 1024, and thus a segment may be about 4MB.

In fact, the segment is a conceptual structure since duplicate chunks are not actually stored and a segment is represented by *a segment recipe* that has the fingerprints of the chunks in it. The segment recipe has the following two roles in the deduplication system: one is to read data chunks during a backup stream recovery and the other is to compare the similarity of segments. In this paper, we focus on the latter role of the segment recipe.

We say two segments are *similar* if they share a number of the same chunks; that is, they have a sufficient number of identical fingerprints in their recipes. For a new segment, its chunks are detected as duplicate or not against those segment recipes of similar segments which are desired to prefetch into cache. Thus a segment recipe serves the basic memory prefetching unit from disk in detecting a chunk duplicate or not. In order to prefetch segment recipes efficiently, the recipe and data chunks of a segment are stored separately in disks.

In order to measure and compare the similarities of segments, a certain number of fingerprints in a segment recipe are sampled to represent the corresponding segment, called *features of a segment*. It usually depends on an efficient fingerprint index structure to query and locate the similar segments with some given features. In this paper, we will build such the index structure called *the context table*, which stores some information of parts of segments. By the reinforcement learning mythology, we aim to identify the most similar segment (called *a champion*) for an incoming segment by exploiting temporal locality and then prefetch several successive segments (called *followers*) by exploiting spatial locality. Therefore the segment-based deduplication is approximate (*i.e.*, some duplicate chunks may not be detected) since it only relies on a limited number of similar segments. This is a trade-off for higher index lookup performance and lower memory overhead.

B. Reinforcement Learning

Reinforcement learning [25], [26] is a goal-directed learning approach from interaction by an agent over time. The agent gains the knowledge by continuous interaction with the dynamic environment to decide which action to achieve its goal. The action decision is typically driven by a positive or negative reinforcement, which is obtained by the agent in form of a *reward*, without relying on exemplary supervision or complete models of the environment. The learning approach uses a formal framework defining the interaction between a learning agent and its environment in terms of states, actions, and rewards. It has the desired characteristics of self-improvement and on-line performance.

In this paper we employ a specific reinforcement learning model called the *K-armed contextual bandits* [27], which is widely used in content recommendation and prefetching [28], [29]. Our observation is that certain segments can be related to one another with the same feature in data backup streams; thus each segment corresponds to an arm and then a feature is limited to at most k segments in the model. Formally, a formal context-bandit algorithm can be described as follows in discrete trials $t = 1, 2, \cdots$.

1) The agent receives an incoming segment s_t and current observation o_t which typically includes the current reward r_t.
2) It then chooses an arm a_t from the set of arms A_t available (at most recent k arms), which is subsequently sent to the environment. Here an arm represents a segment occurred before.
3) The agent receives payoff r_t, at whose expectation depends on both the segment s_t and the arm a_t and then improves its arm-selection strategy with the new observation $o_t + 1$.

The goal of the agent is to obtain rewards as much as possible in the long term. To achieve this goal, it mainly depends on the feedback and the dynamic update during the runtime.

III. Our Approach

In this section, we propose the Learning-based Indexing and Prefetching Approach (called LIPA for short). We will first describe the architecture overview of LIPA. After presenting the whole algorithmic framework, we give the detailed description of the indexing and prefetching mechanisms.

A. Architecture Overview

For an incoming data stream, segments are the basic processing unit in the deduplication algorithm. Fig. 2 shows the architectural overview of the proposed approach, LIPA, which takes a sequence of new segments as the input. Since we focus on the indexing and prefetching mechanisms during the deduplication process, the segmenting method is out of the scope of the paper.

As shown in Fig. 2, LIPA has the following two core components: the context table and the fingerprint cache. *The context table* is to store the run-time context information for fingerprint lookups during the deduplication process. For an incoming segment, a sampled feature is mapped into a set in the table and its segment information is added to one of the segment list. Thus the context table serves the role of the fingerprint index, and maintains the information of a subset of all segments. For an entry of the context table, it has two important attributes: $score$, which indicates the reward of lookup hits on it over time, and $followers$, the number of its successive segments which are to prefetch into the cache once the segment is chosen as a champion. Both the attributes play the most important role in choosing a champion; moreover, they will be updated when getting feedback from the fingerprint cache.

The fingerprint cache mainly stores fingerprints of some segment recipes. To check a chunk is duplicate or not, its fingerprint always needs to be checked in the cache. When a champion is chosen from the context table, it and some of its followers are prefetched into the fingerprint cache. When some fingerprint lookups are hit in the cache, the cache counts the hit. Once a cached segment recipe is selected for eviction, then it will feedback the hits it records to the context table. Note that the context table does not store fingerprints of each segment, but the fingerprint cache does.

According to the reinforcement learning procedure mentioned above, the basic workflow of LIPA can be described briefly as follows.

1) For an incoming segment s_t, its feature f is first obtained by a sampling algorithm; then a mapping relationship between s_t and f by hashing the feature, denoted $f \mapsto s_t$, is stored into the context table. Each entry of the context table corresponds to a fixed feature f and stores at most k associations. In specific, at most k arms share the same feature f. We model such scenario as a multi-armed bandit problem.

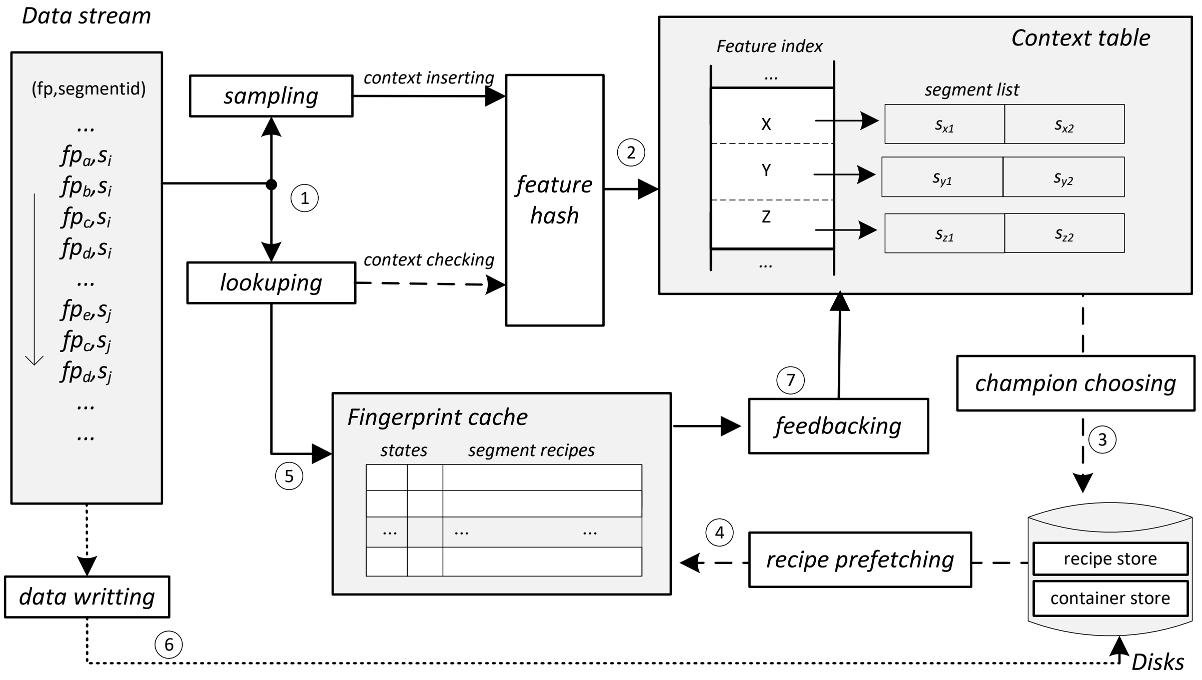

Fig. 2. Architectural overview of the learning-based deduplication method. The marked numbers show the basic workflow and the interplay among the core components, the context table and the fingerprint cache.

2) Observe the current segment s_t and a set A_t of arms together with their related attribute vectors $x_{t,a}$ for $a \in A_t$. The observation in the deduplication is referred to as the context in the k-armed bandit model.
3) From the current context, it makes a decision to choose an arm (*i.e.*, a champion) from A_t by a selection policy. When an arm is selected (*i.e.*, an action is taken), the chosen segment is called a champion.
4) All fingerprints of a champion and its followers are prefetched into memory cache subsequently.
5) For each chunk in segment s_t, its fingerprint is checked against the fingerprint cache to determine if it is a duplicate.
6) After the duplicate check, all non-duplicate chunks and the recipe of the segment are written into disks.
7) If a cached segment is evicted from the fingerprint cache, it feedbacks a reward to the chosen arm and update the context table through the fingerprint lookup hits in cache. During the period of a segment in cache, the hits of fingerprint lookups are recorded. We take the hits as rewards in the K-armed bandit model and further design a reward policy and a feedback mechanism.

Note that these steps are marked in Fig. 2 and the whole process is shown in Algorithm 1. It takes an incoming segment as the input and updates the context table during deduplication. For those non-duplicate chunks, they are written into a write buffer. Once the write buffer becomes full, data chunks are written into disks; at the same time, its recipe is written into the recipe store. Finally, data chunks in the write buffer and its recipe of the input segment are written into disks.

From the algorithm, it can be seen clearly that the deduplication relies on the interaction between of the context table and the fingerprint table. The context table is organized as a key-value structure, depicted in Fig. 3, by which a feature is mapped to a set of segments. In the feedback from the fingerprint cache, the feature and the segment id should be matched before updating the context table.

In the following subsections, we will describe the design and implementation of LIPA in more detail.

B. *Feature Sampling*

After the sequence of chunks in a data stream is segmented by the content defined segmenting method, some fingerprints in every segment are sampled as features. To the similarity between segments, the sparse indexing approach needs to sample a few features of a segment by a given sampling ratio and stores the mapping relationship between features and segments into an indexing structure. It simply uses the uniform sampling method, which selects the first fingerprint from every 2^n fingerprints in a segment (*i.e.*, the sampling ratio $r = 2^n : 1$).

In our approach, we sample only a few features per segment such that the memory cost is reduced remarkably. Suppose we choose ℓ features per segment. The sampling method is to choose ℓ specific fingerprints as features in a segment. As a simple way, the minimum sampling is to choose ℓ minimal fingerprints by a specific comparison rule. Each feature corresponds to an entry in the context table. When more features are sampled in a segment, more champions will be selected, and then more duplicate chunks will be identified.

Algorithm 1 Framework of deduplication algorithm based on the reinforcement learning

Input: A segment of the incoming data stream, seg;
Output: A container buffer, buf.

```
 1: fingerprint $*f$ = sampling_features_of_segment($seg$);
 2: /* lookup the sampled feature in the context table */
 3: if is_in_context_table(t, $f$) then
 4:     /* choose proper segments from recipes into cache */
 5:     fingerprint $*ch$ = choosing_champions($f$);
 6:     cache_prefetch($ch$);
 7:     if evict_recipe_from_cache() then
 8:         feedback_to_context_table();
 9:     end if
10: end if
11: /* check each fingerprint in order */
12: while $seg \neq \emptyset$ do
13:     fingerprint $*f$ = get_fingerprint_from_segment($seg$);
14:     chunk $*data$ = get_chunk_from_segment($seg$);
15:     if is_in_cache($f$) then
16:         /* the chunk is duplicate */
17:         update_to_context_table($f$,$tab$);
18:         write_duplicate_chunk($f$,$buf$);
19:     else
20:         write_unique_chunk($f$, $data$, $buf$);
21:     end if
22: end while
```

Due to the memory overhead constraint, ℓ is often assigned a value no larger than 4 for a segment sized of 1024 (*i.e.*, $r = 256:1$).

C. Champion Choosing

For fingerprint lookups, it is a critical challenge to choose appropriate champions from a set of segments that a feature maps in the context table. As shown in Fig. 3, a feature maps at most k segments. In our design, the context table usually keeps a score for each segment which reflects its contribution to deduplication in the past time; however, there still may have a segment with no score since it has not been chosen before, especially a new one. It does not need to use all of those segments for deduplication, because it is costly to load all of those segments into memory and these segments are much similar due to locality.

There are three policies to choose a champion: *recent, random and ϵ-greedy*. The recent policy always chooses the newest segment as champion and the random policy, as the name implies, always randomly chooses a segment from the candidate set with equal probability, which can be consider the special case of the greedy policy $\epsilon = 0$. In this paper, we employ the common ϵ-greedy policy, which can be considered as the improvement of random policy. In each trial, it may choose a segment with the higher score with $1 - \epsilon$ probability (*i.e.*, exploitation of the past); otherwise, it may choose a random segment with ϵ probability (*i.e.*, exploration of the future). So it is critical to find a better balance between exploitation by selecting a segment known to be useful and exploration by selecting a segment whose usefulness is unknown but which might provide a bigger reward and thus expand the known space.

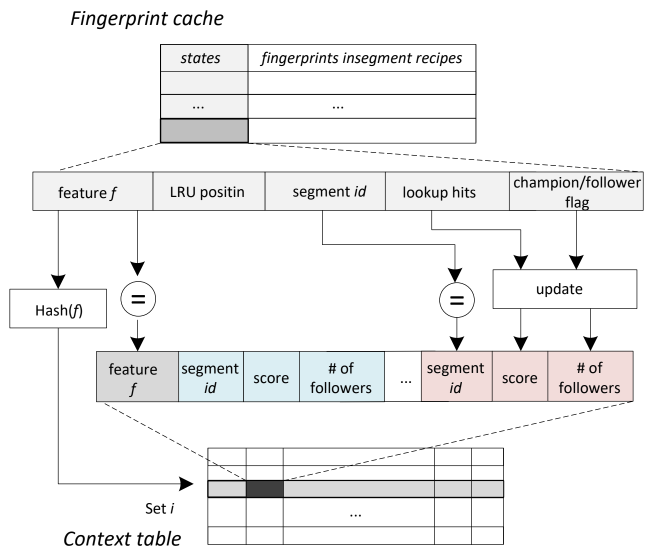

Fig. 3. The structural organization and the interaction of the fingerprint cache and the context table. The *score* and *the number of followers* of a segment in the context table are updated by the feedback of the fingerprint cache according the fingerprint *lookup hits*.

After certain features are sampled in a segment and mapped into the context table, we choose at most m champion candidates. Among the candidates mapped by a given feature, LIPA selects a champion by using the ϵ-greedy policy, which tends to choose one with the highest score with the probability of $1 - \epsilon$. For example, consider the following 4 champion candidates with their respective scores: $(s_1, 0)$, $(s_2, 50)$, $(s_3, 30)$ and $(s_4, 15)$. Besides, segment s_1 is the most recent one with no score, which means that its contribution is unknown to deduplication. The ϵ-greedy policy with $\epsilon = 0.1$ chooses s_2 with probability 0.9 and a random segment with probability 0.1. We will evaluate the ϵ-greedy policy and compare it with the recent policy in the following experiments.

D. Context Table Updating

The updating of the context table is caused by the following two aspects.

1) Entry adding/removing: Recall that the context table keeps the associations between features and the mapped segments. After a feature of a segment is sampled, the association of the feature and the segment is added into the context table as a new entry if the feature is new in the context table; otherwise, the segment is added into the segment queue the feature maps. In each entry of the context table, the number of segments with the same feature in a queue is no more than k. When the queue is full, a new segment is always inserted at the queue tail and existing one has to be removed from the queue.

We consider the following two alternative policies: the *FIFO* policy and the *minimum* policy, to remove an entry in the segment queue. The FIFO policy is to remove an entry in the order of adding into the queue. The minimum policy is to remove the segment with the lowest score. Since k usually is set a small value ($e.g.$, 3 or 4), both policies have no much cost in memory and computation.

2) Reward feedback: Once a champion and its certain number of followers are prefetched into the fingerprint cache, the deduplication looks up fingerprints of the incoming segments. A lookup hit means that a duplicate chuck is identified. During the period of a champion in cache, we add up all lookup hits until it is evicted from cache and then feedback the reward and update the corresponding score in the context table. Let s be a segment and $Q_n(s)$ be the estimated score after segment s is chosen as a champion n times with rewards $r_1, r_2, \cdots, r_n$, respectively. A reward value can be the count of the lookup hits of the segment. One natural way to compute $Q_n(s)$ is by averaging the rewards, *i.e.*,

$$Q_n(s) = \frac{1}{n}\sum_{i=1}^{n} r_i$$

However, a problem of this formula is that the memory and computational requirements grow over time without bound. Instead, it can be implemented in an incremental way,

$$\begin{aligned} Q_n(s) &= \frac{1}{n}\left[(n-1)\,Q_{n-1}(s) + r_n\right] \\ &= Q_{n-1}(s) + \frac{1}{n}\left(r_n - Q_{n-1}(s)\right) \end{aligned}$$

Initially, $Q_0(s) = 0$. This implementation requires memory only for $Q_{n-1}(s)$ and n, and only the small computation for each new reward.

The update process is thus unsupervised, since the training only relies on the feedback that either strengthens or weakens a feature-segment association. A higher reward will strengthen a segment to be a champion. So the feedback is implemented in a lazy manner while the decision to choose the champion can be made in real-time based on current knowledge.

E. Cache prefetching

We observe that similar or identical segments may appear in approximately the same order in a data stream or across multiple streams at a high probability. Such observation calls locality. For example, suppose that the current incoming segment s_i is against the previous segment recipe s_j for data deduplication, then the next incoming segment $s_i + 1$ is to check duplication against the segment $s_j + 1$ with a high probability. By exploiting such locality, we prefetch a certain number of recipes of the champion and its followers into the fingerprint cache after a champion is chosen. It is expected to increase the cache utilization and reduce the accesses to the fingerprint index on disks. The cache uses the least-recently-used (LRU) replacement policy on cached recipes.

During the implementation of LIPA, it is important to adaptively determine the number of prefetched followers once a champion is chosen, denoted by n. *A trivial prefetching way* is to set n to be a fixed value. For example, we prefetch 4 followers at each on-disk access time, namely, $n = 4$. Even though it is easy to implement, the disadvantage is obvious: if n is too small, it may result in too much disk access overhead; otherwise, it may occupy much cache space and decrease cache utilization.

TABLE I
THE CHARACTERISTICS OF FOUR REAL-WORLD DATASETS USED IN OUR EXPERIMENTS.

Dataset name	number of version	Total size	Dupliation ratio
Kernel	115	24.9GB	95.06%
Vmdk	110	180.9GB	42.14%
Fslhomes	126	4.378TB	95.22%
Macos	18	1.628TB	96.69%

As each cached recipe has a reward at feedback, *an adaptive prefetching way* to dynamically adjust the number of followers, n. To implement the adaptive method, the number of prefetched followers is recorded as an attribute of a segment in each entry of the context table, as shown in Fig. 3. Initially, for each new entry, n is set a default value, say 4. Each recipe in the cache records the information including whether it is the last one or not and the champion it follows. When the last recipe is evicted from cache, it will adjust n of its champion according to its reward. Thus most cached recipes tend to be much better for duplication check by the dynamic adjustment. The following experiments will evaluate the performance of LIPA with different default n values and show the performance gains from the dynamic adjustment of n compared with the fixed n.

IV. EXPERIMENTAL EVALUATION

In this section, we first describe the experimental setup on the prototype implementation of LIPA. We then conduct experiments with a number of important parameters in the algorithmic framework with four datasets. It validates the feasibility and effectiveness of LIPA by comparison with the sparse indexing method.

A. Experimental setup

To evaluate the performance of our proposed approach, the prototype system is implemented based on a well-known open source platform called Destor [1], which supports a framework of the deduplication pipelining procedure: chunking, hashing, indexing and storing phases. The deduplication is implemented by multiple threads: each phase corresponds to a thread. We use the following standard algorithms in these phases: the chunking phase uses the TTTD chunking algorithm and the hashing phase uses SHA-1 to calculate fingerprints of chunks. In the implementation of our approach, the segmentation uses the CDS algorithm and the feature sampling uses the minimum sampling method.

The prototype is running in Ubuntu operating system. The hardware configuration includes a quad-core CPU running at 2.4GHz with 4GB RAM and two 1TB 7200rpm hard disks.

We consider the following performance metrics in our experiments.

- Deduplication ratio is defined as the percentage of duplicate data eliminated by the system;
- The RAM usage is the memory overhead for the fingerprint index lookup;
- Data throughput is measured by the deduplicate data amount per second during the backup stream are processed.

Since our paper focuses on a new fingerprint indexing approach, these metrics directly reflect the indexing performance during data deduplication.

B. Datasets

We use four public datasets, which represent different workloads, to evaluate the performance. Table I lists the details of these data sets. In these datasets, Kernel together with Vmdk are the real-world datasets and FSLHomes as well as Macos are data traces.

- Kernel [30] refers to the kernel sources of 155 versions from version 2.3.0 to 2.5.75, collected from the Linux software source server. In this dataset, most files have small sizes; moreover, a source file is usually modified based on the file of the most recent version; that is, the neighboring versions often have the temporal locality property.
- Vmdk [33] is reference to pre-made VM disk images from VMware's Virtual Appliance Market place. We downloaded 110 images of different Linux distributions, such as Ubuntu, CentOS, etc. In this dataset, the size of each image file is considerably large and images of different operating systems have little duplicate data.
- Fslhomes [31] is published by the File system and Storage Lab (FSL) at Stony Brook University. It contains snapshots of students' home directories. The files consist of source code, binaries, office documents, and virtual machine images. We use the trace data of 9 users in 14 days from Sep. 16th to Sep.30th, 2011 to drive our experiments. Every user's backup job is a data stream and thus there are multiple streams every day.
- Macos snapshots [32] also produced by FSL at Stony Brook University, were collected on a Mac OS X Snow Leopard server running in an academic computer lab. We use its snapshots of selected 18 days at the beginning of year 2014.

In our experiments, we take every version of each dataset as a backup job and then averaged all deduplication ratios. The results of four datasets are shown in Table I. For both real-world datasets, Kernel and Vmdk, files are chunked into an average size of 4KB. In addition, the storing phase is to append new chunks in a data stream to a fixed-sized container (4MB). Both trace datasets, Fslhomes and Macos, only contain the content chunking information and the 48-bit hashing fingerprint of each chunk. In our experiments, we only consider the average chunk size of 8KB; the chunking and hashing phases are not measured and only segment recipes of data streams are stored into disks.

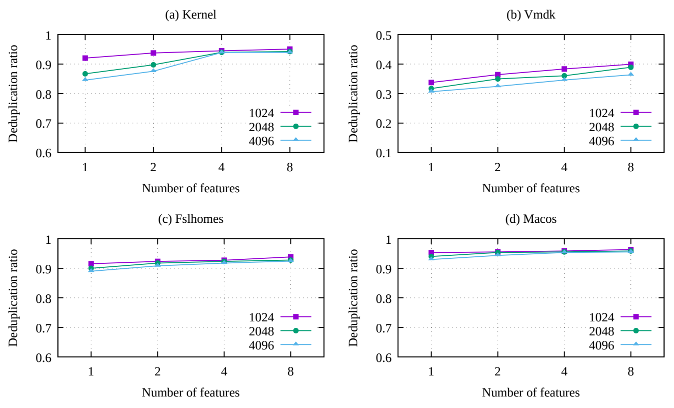

Fig. 4. Deduplication ratios with different feature sampling ratios and segment sizes under four datasets. The deduplication ratio (the y-axis) varies with different numbers of features (the x-axis).

C. Performance Impact of LIPA

As stated in the previous section, there are several alternative policies in the implementation of LIPA and therefore the deduplication performance of LIPA is a comprehensive result of the learning-based indexing and prefetching mechanisms. Before presenting the overall performance, we first understand which aspects of LIPA contribute to deduplication by turning each property on and off from the following aspects.

1) Impact of feature sampling: First, we examine the performance impact of the sampling ratio of a segment. We conduct experiments for three kinds of segments whose sizes are 1024, 2048 and 4096, respectively. For each segment size, we sample different number of features per segment, *i.e.*, ℓ is set to 1, 2, 4 and 8, respectively. Fig. 4 plots the deduplication ratios against various sampling ratios of a segment for the four datasets, Kernel, Vmdk, Fslhomes and Macos.

We have the following observations: *First*, the deduplication ratio achieved increases as the sampling ratio increases and as the segment size increases. In more detail, when the size of each segment is smaller (*i.e.*, the fine-grained segment), it yields a higher deduplication ratio; moreover, the more features are sampled, the higher deduplication ratio achieves. *Second*, it shows the trend that the deduplication could converge to the optimal when the sample number is large enough. It suggests that the sampling numbers determine the deduplication effect, because it depends on the prefetching segments based sampling feature. For the four datasets, the deduplication ratio with 1 sampled feature is much closer to those with $2, 4, 8$ sampled features. So we set the segment size to 1024 and sample $\ell = 1$ feature per segment by default in the following experiments.

2) Impact of the champion choosing: In order to show how the champion choosing policy of LIPA affect the results, we conduct experiments from the following aspects.

Effectiveness of the greedy champion choosing method. In order to clearly show the performance gains from the greedy-based champion choosing method, we use the recent policy rather than the greedy method to choose a champion in our

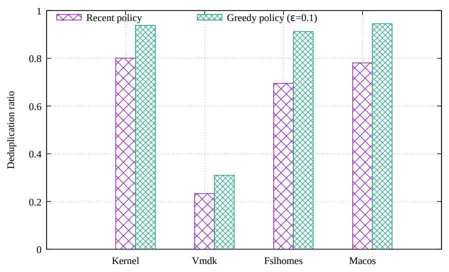

Fig. 5. Deduplication ratio comparison of different champion choosing policies, the recent policy and the greedy policy ($\epsilon = 0.1$) .

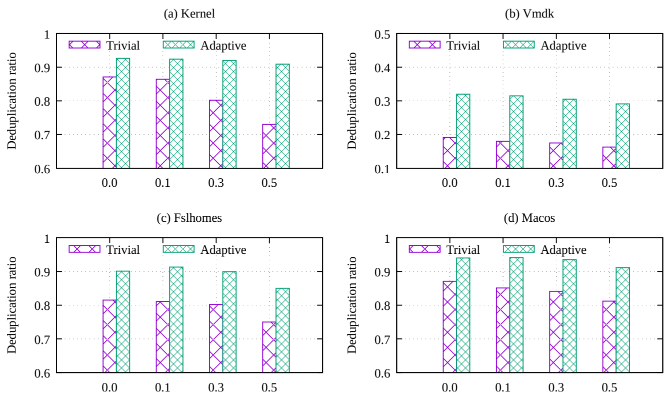

Fig. 6. Deduplication evaluation of the greedy champion choosing method by varying $\epsilon = 0, 0.1, 0.3$ and 0.5, respectively, combined with two different follower prefetching ways. The trivial way always prefetches a fixed number of followers ($n = 4$ here) while the adaptive way prefetches a dynamic number of followers which is initialized by the same n.

experiments. As mentioned in Section III.C, the recent policy always chooses the newest segment as a champion. It means that we turn off the feedback to update the score of each segment recipe in the context table. The results are shows in Fig. 5, compared with LIPA with $\epsilon = 0.1$. The greedy-based method achieves about $10\% \sim 20\%$ over the recent policy under the four datasets.

Impact of the greedy parameter ϵ. We evaluate the impact of the greedy champion choosing method with various ϵ values, $0, 0.1, 0.3$ and 0.5, combining with two different followers prefetching ways, respectively. The results are shown in Fig.6. Note that the greedy champion choosing and the adaptive follower prefetching are the core components in LIPA. Recall that the smaller value of ϵ tends to choose a champion with a higher score. We can clearly observe that the greedy champion choosing policy with a smaller ϵ value results in higher deduplication ratio. So it reveals that the learned score reflects the positive effect on choosing a champion: a segment with a higher score could help detect more duplicate chunks. We also observe that the experiments with $\epsilon = 0.1$ have slightly higher deduplication ratio than those with other ϵ values. It reflects the trade-off between exploration and exploitation of the reinforcement learning to choose a segment and prefetch it into the fingerprint cache. Therefore we set $\epsilon = 0.1$ by default

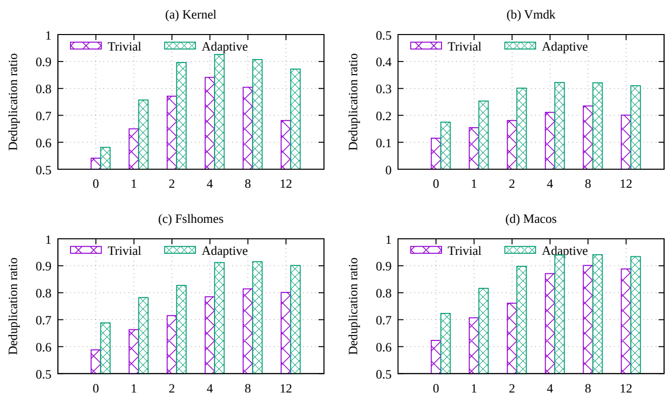

Fig. 7. Deduplication ratio comparison of the trivial prefetching way and the adaptive prefetching way, by varying $n = 0, 1, 2, 4, 8$ and 12, respectively and fixing $\epsilon = 0.1$. The trivial way always prefetches n followers while the adaptive way prefetches a dynamic number of followers which is initialized by the same n.

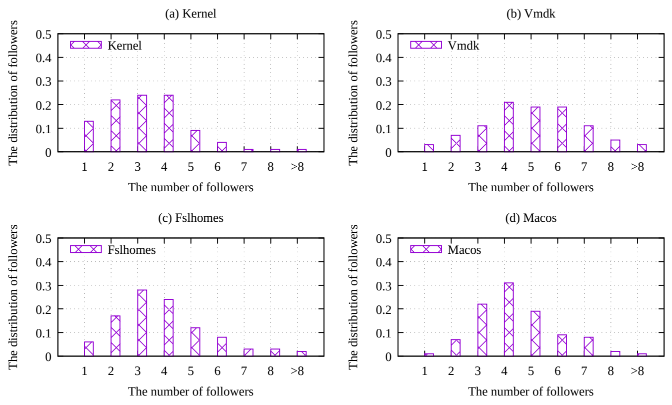

Fig. 8. The distribution of followers (n) of LIPA with $\epsilon = 0.1$, captured by a snapshot of the context table at the end of each experiment. The y-axis shows the ratio of the number of followers varying on the x-axis.

in the following experiments.

3) Impact of the adpative prefetching way: After evaluating the greedy champion choosing policy, we next validate the effect of the adpative prefeching of LIPA by the following experiments.

Effectiveness of the adaptive prefetching. To show the effectiveness of the adaptive prefetching way, we take the following experimental methodology. We replace the adaptive prefetching of LIPA with the trivial prefetching, and then evaluate and compare its performance with LIPA. To make the comparison fair, the initial value of n in the adaptive prefetching way is set the same value as the fixed number of followers in the trivial way. Note that the feedback mechanism of LIPA will dynamically adjust n after the initialization of followers for each new entry in the context table.

Fig. 6 depicts that the adaptive prefetching always achieves higher deduplication ratio than the trivial prefetching with different ϵs and $n = 4$. It reveals that the adaptive prefetching achieves more deduplication gains than the trivial prefetching.

Impact of the default n followers of the adaptive prefetching. We next evaluate the performance influence by the default number of followers (n) for the adaptive prefetching. In our experiments, we consider the following default values of n: $0, 1, 2, 4, 6, 8$ and 12, respectively.

From the experimental results shown in Fig. 7. The results clearly show that the adaptive prefetching has more benefits than the trivial prefetching and also make sense that increasing the number of followers helps deduplication until the value of n is too large. As an important observation, different default n values affect deduplication for the adaptive prefetching of LIPA. If n is too small or too large, it takes much more time to adjust (increase or decrease) the value until it becomes steady. That is, if the default value is too large, some non-useful followers waste some cache space; otherwise, some useful followers are not cached. Therefore, the default value of n influences deduplication performance. It is important to set a proper value for n. From the plots, $n = 4$ may be a proper value for the four datasets.

Distribution of the number of followers. Furthermore, we exam the distribution of the number of followers LIPA prefetches for $n = 4$, as shown in Fig. 8. The experimental results are captured by a run-time snapshot of the context table at the end of each experiment for the four datasets. The y-axis shows the ratio of the number of followers varying on the x-axis. From the results, the distribution of the number of followers has a different skewed shape for each dataset. It may be determined by data access patterns in these datasets. The number of followers can be dynamically adjust during deduplication, and then the prefetched fingerprints in cache tends to be more useful to check duplication.

4) Impact of the context table maintenance: As depicted in Fig. 9, we compare the minimum policy with the FIFO policy for the four datasets. Clearly, the minimum replacement policy achieves better deduplication ratio than the FIFO policy for Fslhomes, Macos and Vmdk; the FIFO replacement policy is better than the minimum policy for Kernel. The reason is that Kernel is composed of series continuous backup versions which consist of many small files; moreover, data chunks have strong dependency on the lasest prior version. So the FIFO policy is better fit to the dataset. Therefore, it reflects an advantage of the reinforcement learning: It is much better to select the appropriate replacement policy for different backup plans with different characteristics in order to achieve a higher deduplication ratio.

D. Comparison Evaluation

Here we present and compare the overall performance of LIPA with the sparse indexing. Considering the sparse indexing has the similar algorithmic skeleton as LIPA, we implement the approach and compare it with LIPA. We evaluate the experimental results of the sparse indexing approach under the same four datasets. The experiments set the following parameters for the sparse indexing: the average segment size is 1024, only one champion chosen and three sampling ratios, $256 : 1$, $128 : 1$ and $64 : 1$, respectively. For an incoming

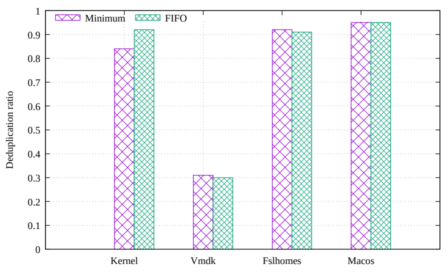

Fig. 9. Deduplication comaprison between the FIFO policy and the minimum policy with LIPA ($\epsilon = 0.1$) under the four datasets.

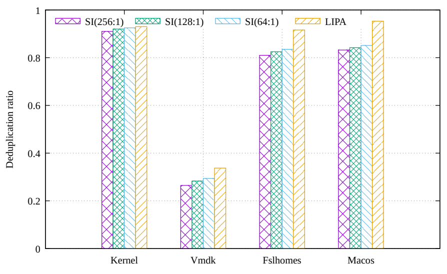

Fig. 10. Comparsion in deduplication ratio between LIPA with $\epsilon = 0.1$ and the sparse indexing with different sampling ratios ($r = 256 : 1, 128 : 1$ and $64 : 1$) under four datasets.

segment, it then chooses a champion which shares the most number of features with the incoming segment from candidates mapped by the features of the incoming segment.

Fig. 10 shows that the comparison of deduplication ratios for the four datasets between LIPA and sparse indexing. The sparse indexing removes $60\% \sim 95\%$ of duplicate data, but LIPA increases the deduplication ratio by $4\% \sim 10\%$ than the sparse indexing. Thus the learning approach is able to enhance the association between fingerprint and segments, and then improves the accuracy of prediction gradually.

The amount of RAM usage required by LIPA and the sparse indexing approach should be linearly proportional to the maximum possible number of unique chunks stored; moreover, the sampling ratio is the primary factor of RAM usage in the experiments. In the implementation, each entry of a sparse index consists of a fingerprint, the address in a container and

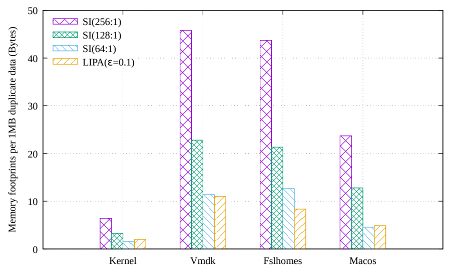

Fig. 11. Comparison in RAM overheads between LIPA with $\epsilon = 0.1$ and the sparse indexing (SI) with different sampling ratios ($r = 256 : 1, 128 : 1$ and $64 : 1$) under four datasets.

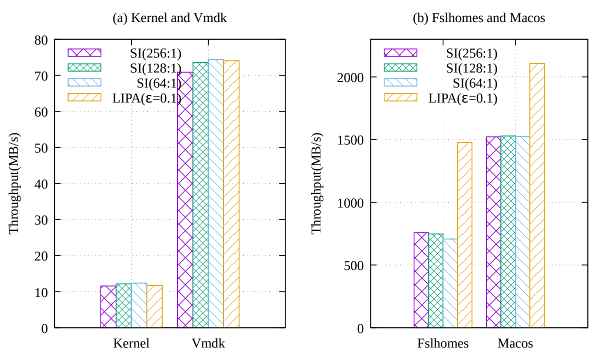

Fig. 12. Comparison in throughput between LIPA with $\epsilon = 0.1$ and the sparse indexing (SI) with different sampling ratios ($r = 256 : 1, 128 : 1$ and $64 : 1$) under four datasets.

the segment address in recipe; besides these fields, each entry of the context table in LIPA has the score and its current hits, 8 bytes. Even though each entry of context table in LIPA has more fields than each entry of the sparse index, LIPA keep much less fingerprints as features than the sparse indexing. Fig. 11 shows the RAM usage of LIPA and sparse indexing with various sampling ratios mentioned above. For example, the sparse indexing with sampling ratio of $128 : 1$ stores 4 fingerprints per segment as represented fingerprints whereas LIPA stores only one feature per segment; in such situation, the sparse indexing is about 4 times the memory usage of LIPA for the four workloads. For the other sample ratios, LIPA also has less memory usage than the sparse indexing.

We further examine the throughput in data deduplication under the four datasets. Fig. 12 shows a comparison of data throughput between LIPA and the sparse indexing. We observe that LIPA and the sparse indexing have almost the same throughput for Kernel and Vmdk since Kernel is dominated by small files and Vmdk has much less redundant data to remove. However, for the other two datasets, LIPA achieves much better throughput than sparse the indexing on Fslhomes and Macos. It can reduce to the following reasons: one is that LIPA reduces the accesses of on-disk index and the other is that much more duplicate data identified doesn't need to store.

In summary, the comprehensive comparison shows that LIPA only requires little memory overheads to store the index but achieves the same or even better deduplication ratio and throughput than the sparse indexing.

E. Discussion

We consider the benefit of LIPA comes from the feedback of the fingerprint cache, because it dynamically adjusts two properties of a segment in the context table: *score* and *number of followers*. The former represents the number of fingerprint lookup hits in the corresponding segment recipe. So it is helpful to identify champions for an incoming segment by exploring temporal locality. The latter represents the number of successive segment recipes which will be prefetched into the fingerprint cache next time with the champion. If more fingerprint lookups are hit in followers of a champion, the feedback increase the number of the followers; otherwise, the feedback decrease the number of the followers. Thus most segment recipes in the fingerprint cache are useful to detect duplicate chunks in the fingerprint cache by exploring spatial locality. As a comparison, some previous methods often prefetch a fixed number of segment recipes every time. So the main idea behind LIPA is to exploit both temporal locality and spatial locality by designing the learning-based indexing and prefetching mechanisms.

In the process of deduplication, imagine that the following extreme case: the scores of entries in context table may be nearly equal to each other. In the situation, LIPA would become invalid since the chosen features have no discrimination, which degrades to the sparse indexing deduplication. So we argue that our proposed algorithm is the generalization of the similarity-based deduplication. In fact, we don't need worry about the degradation. As revealed of the data analysis in [15], the skewed chunk popularity and access patterns of users' data access behaviors require the deduplication algorithm more intelligent and adaptive. In the future work, more complex models of bandit rewards may be worth studying. We will strengthen LIPA to capture the regular and irregular patterns by using a number of upper-level access hints and file attributes.

V. Related Work

Deduplication has become inevitable components in both archival and primary storage for data centers and cloud storage like infrastructure [6], [22], [12], [34], [35]. As the amount of the stored data grows in these large-scale systems, the total size of the fingerprints representing data chunks becomes beyond the memory capability. Consequently, how to efficiently store and index these fingerprints of data chunks stored on the disks becomes a severe performance bottleneck in these large-scale data systems. Such a problem is called the fingerprint indexing problem and is challenging[1], [3], [2]. There are much related work to speedup deduplication from the following dimensions.

A. Trading off between storage space and performance

As one important means, the data structure of fingerprint index directly determines the deduplication ratio. In recent years, a lot of work has tried to propose efficient fingerprint-indexing schemes to improve the deduplication performance by different ways. As a milestone work, the Data Domain File System (DDFS) [17] uses a Bloom filter, stream-informed segment layout, and a locality preserved cache, together reducing the disk I/O for fingerprint index lookup. Furthermore, the sparse indexing [18] improves DDFS memory utilization by sampling the index of chunk fingerprints in memory, instead of using Bloom filters as in DDFS, which reduces the RAM usage to less than half of that in DDFS. Since then, many deduplication approaches, e.g., Silo [19], [20], have been proposed and deduplication performance has been improved significantly [9], [19], [11], [12].

For some state-of-the-art approaches, a fingerprint indexing scheme usually can lead to either the exact deduplication or the approximate deduplication [1], [2]. The exact deduplication means that all duplicate chunks are eliminated [17], [9], [11]; however, it is too time-consuming to lookup a fingerprint and too inefficient to restore data. In the recent work, the exact deduplication often improves the lookup performance by utilizes the emerging disk devices. For example, J. Ma et al. [13] propose a lazy deduplication scheme by performing on-disk lookups in batches in SSD disks.

As opposed to the exact deduplication, the approximate deduplication trades a slightly reduced accuracy of duplicate detection for a higher index-lookup performance and lower memory overhead. This family of methods includes sparse indexing [18] and Silo [19], [20]. It does not search for uncached fingerprints on disk and thus a small number of duplicate chunks are not detected. Thus it is an essential sacrifice to boost the deduplication performance. Nowadays such the approximate methods can be potentially benefit to speedup the fingerprint index lookup and save memory footprints and hence it becomes more prevalent than the exact method.

B. *Exploring data locality*

As the other important meanings, locality is important to speedup the deduplication. When a fingerprint is found on disk, prefetching is invoked, whereby adjacent fingerprints stored on disk are transferred to the cache. Deduplication systems take advantage of locality properties in data streams to reduce disk accesses by using cache [23]. DDFS [17] captures locality by storing and prefetching in the order of the stored unique chunks in containers. As fingerprints frequently arrive in the same order as they arrived previously and therefore the same order as they are stored on the disk, this prefetching strategy leads to a higher hit rate in the fingerprint cache which significantly reduces disk access time. MAD2 [8] employs a Bloom filter array as a quick index for deduplication and also preserves locality of fingerprints in cache. In [19], it exploits the inherent locality of the backup stream with a progressive sampled indexing approach. Block locality caching (BLC) [11] improves indexing performance by exploiting the locality of the most recent backup in a long-term backup system. From all of these methods, Fu et al. [1] comprehensively study and evaluate some locality-based and similarity-based approaches; moreover, they distinguish the locality into two categories, logical and physical locality.

C. *Exploring access patterns*

Nowadays much work has recognized the important role of analyzing characteristics and patterns of real-world data sets and expects to help inform future deduplication storage designs.

To utilize the deduplication context information, SAM [9] exploits file semantics of size, type, locality, etc., to optimize fingerprint indexing. As the work of Silo [19], [20] and RMD [12], the researchers explore the similar segment detection and the access locality becomes critical since both supply a larger query range to determine a duplicate chunk or not. Similar segments may be distributed among different locations, which incurs multiple I/Os in order to access all similiar segments or less I/Os only accessing part of these similar segments. However, these methods are lack of the feedback mechanism in the long-term system evolution. HANDS [14] dynamically pre-fetches fingerprints from disk into cache according to working sets statistically derived from access patterns. In recent literature [15], they gave a long-term user-centric analysis of deduplication patterns with a data set that spans a period of 2.5 years. For the frequent virtual machine snapshot backups, D. Agun et al. [6] present a source-side backup scheme by using popular data chunks shared among snapshots.

All of these methods illustrate the feasibility by using the data analysis way, which motivates us to apply the learning method in deduplication. The main diffference to these methods, LIPA has a feedback scheme by exploring the reinforcement learning [24], [25]. In essence, the reinforcement learning is applicable to identify access patterns in the long-term access evolution [28], [29] . In this paper, we employ a specific reinforcement learning model called contextual bandits [26]. Our work is the first attempt of reinforcement learing in deduplication and achieves the desired effect.

VI. Conclusions

In this paper, we propose fingerprint index detection and prefetching technique based on reinforcement learning to solve disk bottleneck problem. We update the association relationship between features and their corresponding segments by a feedback mechanism, and dynamically adjust the cache mechanism to improve the performance of deduplication system. Experimental results show that our learning based method only requires little memory overheads to store the index but achieves the same or even better deduplication ratio than previous methods. It can remove $2\% \sim 10\%$ more redundant data than sparse indexing. On the memory footprints for different datasets, the learning method can save $2 \sim 20$ times than sparse indexing. Therefore, the experimental results validate that our proposed learning-based algorithm can effectively and efficiently predict the future neighborhood and adaptively improve the prediction accuracy.

Acknowledgment

We appreciate it that Philip Shilane could be our shepherd and thank the anonymous reviewers for their constructive comments for our paper. We are grateful to Prof. Gang Wang who gave some valuable helps on the work.

The work was partly supported by NSFC project (No.61201234 and 61202381) and NSF project of Tianjin (No. 17JCYBJC15600 and No.18JCTPJC48700).

References

[1] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, ”Design tradeoffs for data deduplication performance in backup workloads,” in Proc. USENIX Conference on File and Storage Technologies (FAST’15). Santa Clara, CA, USA, 2015, pp.331-344.

[2] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," in Proceedings of the IEEE, vol. 104, no. 9, pp. 1681-1710, 2016.

[3] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," in ACM Computing Surveys (CSUR), vol.47, no.1, p.1-30, 2014.

[4] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low bandwidth network file system", in Proc. ACM Symposium on Operating Systems Principles (SOSP'01), Banff, Canada, 2001.

[5] C. Policroniades and I. Pratt, "Alternatives for detecting redundancy in storage systems data," in Proc. USENIX ATC'04, 2004, pp.73-86.

[6] D. Agun, T. Yang, "Low-Profile Source-side Deduplication for Virtual Machine Backup," in Proc. HotCloud, 2016.

[7] D. Meister and A. Brinkmann, "Multi-level comparison of data deduplication in a backup scenario", in Proc. ACM SYSTOR 2009, Haifa, Israel, May 2009.

[8] J. Wei et al., "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in Proc. IEEE 26th Symp. Mass Storage Syst. Technol., May 2010, pp. 1-14.

[9] Y. Tan et al., "SAM: A semantic-aware multi-tiered source de-duplication framework for cloud backup," in Proc. 39th Int. Conf. Parallel Process (ICPP'10), Sep. 2010, pp. 614-623.

[10] F. Guo, P. Efstathopoulos, "Building a high performance deduplication system", in Proc. USENIX ATC, 2011.

[11] D. Meister, J. Kaiser, and A. Brinkmann, "Block locality caching for data deduplication," in Proc. 6th Int. Syst. Storage Conf., Jun. 2013, pp. 1-12.

[12] P. Zhang et al., "RMD: A Resemblance and Mergence based Approach for High Performance Deduplication", in Proc. International Conference on Parallel Processing (ICPP), 2016, pp. 536-541.

[13] J. Ma, R. J. Stones, Y. Ma, J. Wang, J. Ren, G. Wang, X. Liu, "Lazy Exact Deduplication", in Proc. the 32nd International Conference on Massive Storage Systems and Technology (MSST 2016), 2016.

[14] A Wildani, E. L. Miller, O. Rodeh, "HANDS: A heuristically arranged non-backup in-line deduplication system," in Proc. IEEE International Conference on Data Engineering (ICDE 2013), 2013, pp.446-457.

[15] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, et al., "A long-term user-centric analysis of deduplication patterns," in Proc. the 32nd International Conference on Massive Storage Systems and Technology (MSST 2016), 2016.

[16] S. Quinlan, S. Dorward, "Venti: a new approach to archival storage," in Proc. USENIX FAST, 2002.

[17] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in Proc. USENIX FAST, 2008.

[18] M. Lillibridge, K. Eshghi, D. Bhagwat et al., "Sparse indexing: Large scale, inline deduplication using sampling and locality." in Proc. USENIX FAST, 2009.

[19] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput," in Proc. USENIX ATC, 2011, pp. 285-298.

[20] W. Xia, H. Jiang, D. Feng, and Y. Hua, "Similarity and locality based indexing for high performance data deduplication," in IEEE Transactions on Computers, vol. 64, no. 4, pp. 1162-1176, 2015

[21] D. Bhagwat, K. Eshghi, D. D. Long et al., "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in Proc. IEEE MASCOTS, 2009.

[22] W. Dong, F. Douglis, K .Li, H. Patterson and P. Shillane, "Tradeoffs in scalable data routing for deduplication clusters," in Proc. USENIX FAST, 2011.

[23] Y. Oyama, J. Murakami, S. Ishiguro et al, "Implementation of a deduplication cache mechanism using content-defined chunking," in International Journal of High Performance Computing & Networking, vol. 9, no. 3, pp.190-205, 2006.

[24] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: a survey," in Journal of Artificial Intelligence Research, vol. 4, pp. 237-285, 1996.

[25] R. S. Sutton and A. G. Barto, "Reinforcement Learning: An Introduction," Cambridge, MA, USA: MIT Press, 1998.

[26] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The non-stochastic multi armed bandit problem," in SIAM Journal on Computing, vol. 32, no. 1, pp.48-77, 2002.

[27] L. Zhou, "A survey on contextual multi-armed bandits", arXiv preprint arXiv:1508.03326.

[28] L. Li, W. Chu, J. Langford, and R. E. Schapire, "A contextual bandit approach to personalized news article recommendation," in Proc. ACM WWW, 2010.

[29] L. Peled, S. Mannor, U. Weiser, Y. Etsion, "Semantic Locality and Context-based Prefetching Using Reinforcement Learning," in Proc. ACM ISCA, 2015.

[30] Linux kernel. http://www.kernel.org/.

[31] FSLhomes. http://tracer.filesystems.org/traces/fslhomes/.

[32] MacOS. http://tracer.filesystems.org/traces/macos/

[33] Vmdk. http://www.thoughtpolice.co.uk/vmware/.

[34] Yucheng Zhang, Dan Feng, Hong Jiang, Wen Xia, Min Fu, Fangting Huang, Yukun Zhou, "A Fast Asymmetric Extremum Content Defined Chunking Algorithm for Data Deduplication in Backup Storage Systems", IEEE Transactions on Computers (TC), Vol. 66, No. 2, pp. 199 - 211, 2017.

[35] H. Wu, C. Wang, Y. Fu, S. Sakr, K. Lu, L. Zhu, "A Differentiated Caching Mechanism to Enable Primary Storage Deduplication in Clouds," in IEEE Transactions on Parallel and Distributed Systems, 2018, doi: 10.1109/TPDS.2018.2790946.