DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression

Jisung Park, *ETH Zürich;* Jeonggyun Kim, Yeseong Kim, and Sungjin Lee, *DGIST;* Onur Mutlu, *ETH Zürich*

https://www.usenix.org/conference/fast22/presentation/park

This paper is included in the Proceedings of the 20th USENIX Conference on File and Storage Technologies.

February 22–24, 2022 • Santa Clara, CA, USA

DeepSketch: A New Machine Learning-Based Reference Search Technique for Post-Deduplication Delta Compression

Jisung Park[1*] Jeonggyun Kim[2*] Yeseong Kim[2] Sungjin Lee[2] Onur Mutlu[1]

[1]*ETH Zürich*   [2]*DGIST*

Abstract

Data reduction in storage systems is becoming increasingly important as an effective solution to minimize the management cost of a data center. To maximize data-reduction efficiency, existing post-deduplication delta-compression techniques perform delta compression along with traditional data deduplication and lossless compression. Unfortunately, we observe that existing techniques achieve significantly lower data-reduction ratios than the optimal due to their limited accuracy in identifying similar data blocks.

In this paper, we propose *DeepSketch*, a new reference search technique for post-deduplication delta compression that leverages the learning-to-hash method to achieve higher accuracy in reference search for delta compression, thereby improving data-reduction efficiency. DeepSketch uses a deep neural network to extract a data block's *sketch*, *i.e.,* to create an approximate data signature of the block that can preserve similarity with other blocks. Our evaluation using eleven real-world workloads shows that DeepSketch improves the data-reduction ratio by up to 33% (21% on average) over a state-of-the-art post-deduplication delta-compression technique.

1 Introduction

As modern data centers generate a tremendous volume of new data every day, it becomes critical for sustainability to store such large amounts of data in an economical way. Employing a data-reduction technique is one of the effective solutions to cut down the management cost of a data center. A data-reduction technique reduces the amount of data physically stored in storage media by reducing data redundancy, which allows a data center to handle the same amount of data with fewer or smaller resources (*e.g.,* storage devices and servers).

Many prior works have proposed various data-reduction techniques based on data compression [9,30,46,51,52,58] and data deduplication [12,21,22,26,36,49,59,62,67,76,88]. Data compression encodes a data block using lossless-compression algorithms so that a smaller number of bits can represent the data block. Data deduplication prevents a data block from being written if there already exists an *identical* data block (*i.e.,* a block that contains exactly the same data) in the storage system. To achieve a high data-reduction ratio (*i.e., Original Data Size / Reduced Data Size*), some studies [45,55] integrate the two techniques in a manner that first applies data deduplication for incoming (*i.e.,* to-be-stored) blocks and performs lossless compression on non-deduplicated blocks.

Delta compression [3, 8, 64, 75, 81, 82, 86] has recently received increasing attention as a complementary method to overcome the limitations of data compression and data deduplication. It compares the data block to compress with a *reference* data block and extracts only different bit patterns between the two blocks, which are then encoded using lossless compression. The more similar the data block and the reference (*i.e.,* the smaller the delta between the data and the reference), the higher the data-reduction ratio. By leveraging the similarity between two blocks, delta compression can achieve a high data-reduction ratio even for non-duplicate data (which cannot benefit from data deduplication) and high-entropy data (which lossless compression cannot efficiently handle). Several prior works [64, 75, 82, 86] demonstrate that *post-deduplication delta compression*, which performs deduplication, delta compression, and lossless compression in order, can significantly improve the data-reduction ratio over simple integration of deduplication and lossless compression.

A key challenge for post-deduplication delta-compression techniques is how to find a good reference block that provides a high data-reduction ratio. The most intuitive approach is to scan all the data blocks stored in the storage system and use the one that provides the highest data-reduction ratio as the reference for the incoming block. Unfortunately, doing so is practically infeasible due to its prohibitive performance overhead. To address this, prior works [64, 75, 82, 86] use *locality-sensitive hash (LSH)* functions [7, 34] to generate similar data signatures for data blocks with similar bit patterns, which is called *data sketching*. Data sketching enables quick reference search across a large-scale storage system by comparing only the signatures (*i.e.,* sketches) of data blocks.

In this work, we observe that existing post-deduplication delta-compression techniques [75, 86] achieve significantly lower data-reduction ratios than the optimal due to the high *false-negative rate (FNR)* of LSH-based reference search. Our analysis using six real-world workloads shows that, although a state-of-the-art reference search technique [86] is effective at identifying a *very similar* reference block (which thus provides a very high data-reduction ratio) for an incoming block, it also fails to find *any* reference block for a large number of incoming blocks (up to 75.5%) that actually have at least one good reference block within the storage system. Tuning the used LSH function may be able to reduce the high FNR in reference search, but it would require significant human effort to identify the best settings for each workload.

Our goal is to improve the space efficiency of a storage system by increasing the accuracy of reference search in post-deduplcation delta compression, thereby reducing the gap between existing data-reduction techniques and the optimal.[1] To this end, we propose *DeepSketch*, a

---

[1]In this work, we focus on data reduction rather than other optimizations (*e.g.,* mitigation of performance/memory overheads), targeting systems where space efficiency is the highest priority (*e.g.,* archival or backup systems).

---

*J. Park and J. Kim are co-primary authors.

new machine learning (ML)-based data sketching mechanism specialized for reference search in delta compression. **Our key idea** is to use the *learning-to-hash* method [43, 80] to automatically generate similar data signatures for any two data blocks that would provide a high data-reduction ratio when delta-compressed relative to each other.

For each incoming data block, DeepSketch generates the block's sketch using a deep neural network (DNN) model. It performs DNN inference with the target data block as an input of the DNN and uses the resulting activation values in the DNN's last hidden layer as the data block's sketch. We envision that DeepSketch's DNN is *pre-trained* before building or updating a DeepSketch-enabled system, using data sets collected from other existing systems that store similar (or the same) types of data.

While many prior works [10, 11, 48, 50, 70, 89] demonstrate the high effectiveness of the learning-to-hash method in various nearest-neighbor search applications (*e.g.,* image recognition and classification), applying the learning-to-hash method to the reference search problem in post-deduplication delta compression is not straightforward. A key problem is that, unlike existing ML-based applications that deal with specific known data types (*e.g.,* images and voices), DeepSketch needs to process *general binary data*, which introduces two key challenges. First, there is no well-defined labeled data or semantic information (*e.g.,* cats, dogs, and monkeys in image classification) within our target data sets. Second, possible bit patterns of a data block have an extremely high dimensional space (*e.g.,* $2^{4,096 \times 8}$ unique bit patterns for a 4-KiB data block). Due to the high dimensionality of the target data set, it is difficult to collect large enough data to train the DNN with high inference accuracy using known training methods.

To address the above challenges, we develop a new method to train the DNN of DeepSketch, which generates hash values suitable for reference search in post-deduplication delta compression. We extend the traditional unsupervised learning approach [29] in three ways. First, based on the k-means clustering algorithm [53], we design a new clustering method, called *dynamic k-means clustering (DK-Clustering)*, which effectively classifies high-dimensional data without any knowledge of the number of clusters. Second, after clustering, we ensure each cluster to have a sufficient number of data blocks by adding data blocks slightly and randomly modified from each cluster's representative block. Doing so prevents DNN training from being biased towards some specific data patterns that occur frequently. Third, we perform two-stage DNN training to enable DeepSketch to generate a data block's hash value. We first train a DNN to classify data blocks into the clusters formed by DK-Clustering and then transfer the knowledge of the trained DNN to build the learning-to-hash network that generates the hash values (*i.e.,* sketches) of data blocks.

We integrate our DeepSketch engine into a state-of-the-art post-deduplication delta-compression technique [86]. Unlike existing techniques that aim to find a reference block whose sketch *exactly* matches that of the incoming block, we exploit a state-of-the-art *approximate* nearest-neighbor search

algorithm [16]. Doing so allows DeepSketch to tolerate small differences within data sketches (i.e., it can identify similar blocks even when the blocks' sketches are different), thereby increasing the chance of delta compression for an incoming data block. Our evaluation using eleven real-world workloads shows that DeepSketch improves the data-reduction ratio by up to 33% (21% on average) over the state-of-the-art baseline.

This paper makes the following key contributions:
- We propose DeepSketch, the first machine learning-based reference search technique for post-deduplication delta compression. We demonstrate that the learning-to-hash method can be an effective solution to generate delta-compression-aware data signatures for general binary data.
- We introduce a new training method that allows DeepSketch to learn delta-compression-aware data representation for an extremely high-dimensional data set.
- We integrate DeepSketch into the state-of-the-art post-deduplication delta-compression technique [86]. Evaluation results using eleven real-world workloads show that DeepSketch improves the data-reduction ratio by up to 33% (21% on average) compared to the state-of-the-art baseline.

2 Background

We provide brief background on data-reduction techniques in storage systems necessary to understand the rest of the paper.

2.1 Data Reduction in Storage Systems

There are three major data-reduction approaches: 1) data deduplication, 2) lossless compression, and 3) delta compression.
Data Deduplication. Data deduplication [12, 21, 22, 26, 36, 49, 59, 62, 67, 76, 88] reduces the amount of data physically written to storage devices by eliminating duplicate data in the storage system. In data deduplication, an incoming data block is not physically written if it has the same data content as a data block previously stored in the storage system. Instead, the storage system maintains a table that stores mapping information between such a deduplicated block and the previously-stored block with the same content (called *reference*), so that future reads to any deduplicated blocks can be serviced from their reference. This mechanism allows data deduplication to store only a single copy of any block-granularity unique data content in the storage system.

To quickly identify an incoming block's reference, deduplication uses a strong hash function (*e.g.,* SHA1 [78] or MD5 [69]) to generate a data block's *unique* signature, commonly called a *fingerprint*. Given two blocks, deduplication determines whether or not they have the same content, by comparing only the two blocks' fingerprints. To avoid any data loss due to hash collision, it is common practice for deduplication to use a strong hash function to generate fingerprints whose collision rate is lower than the uncorrectable bit-error rate (UBER) requirement of a disk (*e.g.,* $< 10^{-15}$ to 10^{-16} [17, 25, 26, 67]).
Lossless Compression. Data compression [31, 74, 90] is a fundamental method to reduce the size of data in computing systems. Given a data block, it encodes the block's content to

be represented by a smaller number of bits in a manner that replaces repetitive bit patterns with smaller metadata or symbols. Doing so results in an increase in the *entropy* [74] of the compressed data. For a data block with low entropy (*i.e.,* the block contains many repeated bit patterns), lossless compression can achieve a high *data-reduction ratio* (*i.e., Original Data Size / Compressed Data Size*).

Delta Compression. Delta compression [3,8,64,75,81,82,86] eliminates redundant bit patterns that coexist in two different blocks. It stores only either of the two blocks and the difference (*i.e., delta*) between the two blocks. Leveraging the *similarity* of two different data blocks enables delta compression to achieve higher data reduction over 1) deduplication, which removes only *identical* data blocks, and 2) lossless compression, which eliminates redundancy only *within a block* and does *not* work well with high-entropy data. For this reason, delta compression has gained increasing attention in recent studies [64,75,82,86] as a complementary method bridging deduplication and lossless compression.

A key challenge for delta compression in large-capacity storage systems is how to find a good reference block that provides a high data-reduction ratio for each incoming block. Designing a reference search technique for delta compression is similar to solving a nearest-neighbor search problem, as its goal is to find the most similar data block (which does not have to be an *exact* match) within a large data set for a given incoming block. The most widely-used approach is to use locality sensitive hashing (LSH) [7,34] to generate a *sketch* of a block [64,75,82,86], which is a *more approximate* signature than the block's fingerprint (used in deduplication for exact-match searching). An LSH function $L(d_i)$ is designed to hash data d_i, such that the more similar the given data d_1 and d_2, the lower the bit-pattern difference between $L(d_1)$ and $L(d_2)$. LSH-based data sketching enables quick reference search by comparing only the sketches of data blocks.

2.2 Combined Data-Reduction Technique

For systems where storage efficiency is the paramount requirement, prior works propose to combine the three major data-reduction techniques, called *post-deduplication delta compression*. Figure 1 depicts the overall architecture of a storage system that adopts post-deduplication delta compression [75, 86] to perform deduplication (❶–❸), delta compression (❹–❼), and lossless compression (❽) in order. A data-reduction module (DRM), which can be implemented as an intermediate layer between a file system and storage devices, performs post-deduplication delta compression for a write request to reduce its size. For a read request, it looks for the location of the corresponding compressed data in storage devices and returns the decompressed data. To this end, the DRM maintains a fingerprint (FP) store and a sketch (SK) store for quick reference search for deduplication and delta compression, respectively, along with a reference (Ref.) table to store the mapping information between each deduplicated or delta-compressed block and its reference block.

For each incoming data block, the DRM ❶ first checks

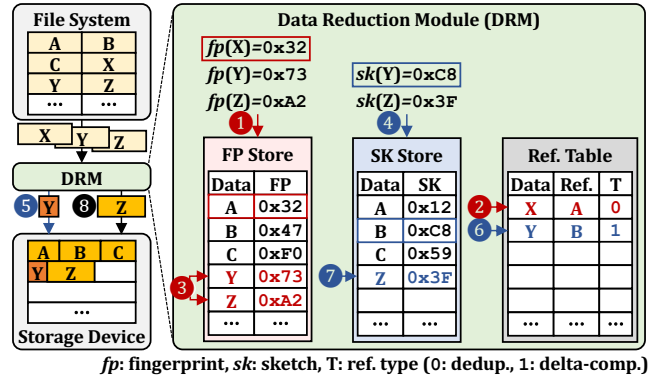*fp*: fingerprint, *sk*: sketch, T: ref. type (0: dedup., 1: delta-comp.)

Figure 1: Overview of post-deduplication delta compression.

if the storage system already contains a data block with the same content by referring to the FP store. If the incoming block's fingerprint matches one in the FP store (*e.g.,* block X matching block A in Figure 1), the DRM skips writing the block to the storage device and ❷ just updates its mapping information in the reference table in order to redirect future reads for the incoming block to the matching reference block. To use non-deduplicated blocks (*e.g.,* blocks Y and Z) as potential reference data for deduplication in the future, the DRM ❸ writes their fingerprints into the FP store.

If there is no matching fingerprint in the FP store, the DRM ❹ searches for matching sketches in the SK store to find a reference block for delta compression. When it finds a reference block (*e.g.,* block B for block Y), the DRM ❺ performs delta compression with the reference and stores the compressed data. There is a possibility of having multiple matching references in the SK store (see Section 3.1). In such a case, the DRM usually selects the first-found candidate (called *first-fit* selection) [75, 86]. The DRM then ❻ updates the reference table to map the incoming block to the reference block so that it can decompress the delta-compressed data using the reference block for future read requests. If no matching sketch is found in the SK store (*e.g.,* block Z), the DRM ❼ adds the incoming block's sketch into the SK store to use the incoming block as a potential reference block for delta compression in the future. Finally, the DRM ❽ compresses the block with a lossless compression algorithm and stores the result.

3 Motivation

In this section, we discuss 1) the limitations of existing LSH-based post-deduplication delta-compression techniques [75, 86] and 2) the potential of the learning-to-hash method [43,80] for more accurate reference search in delta compression.

3.1 Limitations of LSH-Based Sketching

As explained in Section 2.1, LSH-based data sketching enables quick search for a reference block (*i.e.,* reference search) in post-deduplication delta compression. Figure 2 describes the high-level idea of state-of-the-art LSH-based sketching schemes [75, 86], which we call *super-feature data sketching (SFSketch)*. SFSketch generates a data block's sketch using *m features* extracted from the block (*e.g., m* = 12 in Figure 2). To extract a feature $F_i(A)$ of block A ($0 \leq i \leq m-1$), as shown in
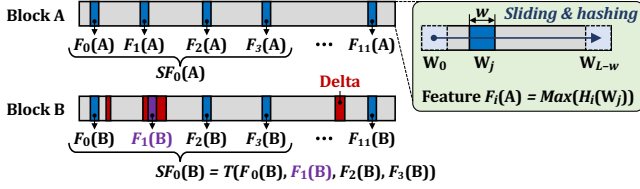
Figure 2: An example of LSH-based sketching.

Figure 2 (right), SFSketch calculates the hash value $H_i(W_j)$ of each sliding window W_j, where j is the starting byte position of the window in the block. Given a block size of L and a window size of w, $(L-w+1)$ hash values are calculated in total, and the maximum hash value $Max(H_i(W_j))$ is selected as feature $F_i(A)$. SFSketch repeats this process to extract m features using a different hash function for each feature (*i.e.*, H_i for F_i) and then builds N super-features (SFs) by transposing the m features (*e.g.*, given $m=12$ and $N=3$, $SF_k(A) = T(F_{4k}(A)$, $F_{4k+1}(A)$, $F_{4k+2}(A)$, $F_{4k+3}(A))$, where $0 \leq k \leq 2$).

Using multiple SFs as the sketch of a data block enables SFSketch to tune the accuracy of reference search by changing the *matching criteria* (i.e., criteria for judging the similarity of given two blocks). Consider the example of Figure 2 where block A's data content is almost the same as block B, except for the red regions marked as *Delta*. Suppose that, due to the small differences between blocks A and B, every hash function H_i other than H_1 has the same maximum value $Max(H_i(W_j))$ for blocks A and B, *i.e.*, every feature F_i except F_1 is identical between blocks A and B. In such a case, $SF_0(A)$ does not match $SF_0(B)$ ($\because F_1(A) \neq F_1(B)$), while all the other SFs are identical between blocks A and B. The two blocks can be considered either similar or not depending on the matching criteria; SFSketch may either decide that the two blocks are dissimilar because there exists a different SF or judge that block A resembles block B since their other two SFs (*i.e.*, SF_1 and SF_2) match each other. There are many possible matching criteria, but to maximize the data-reduction ratio, existing SFSketch-based techniques [75, 86] consider that two blocks are similar if there exists at least one matching SF.

While existing SFSketch-based delta-compression techniques provide significant improvement in data reduction compared to a simple combination of deduplication and lossless compression [75, 86], we observe that SFSketch-based reference search often fails to identify a good reference block that can provide a high data-reduction ratio for an incoming block. To show this, we compare a state-of-the-art SFSketch-based reference search technique [86] to *brute-force* search that performs delta compression of an incoming block with *every* stored block and selects the stored block that provides the highest data-reduction ratio as the incoming block's reference.[2] For our evaluation, we use 4,090,975 4-KiB data blocks collected from six different workloads in real systems (see Section 5.1 for our evaluation methodology and workloads).

We use two major metrics to evaluate the accuracy of SFSketch compared to brute-force search: 1) *false-negative rate (FNR)*, the probability of identifying no reference block for an incoming data block even though brute-force search can find one, and 2) *false-positive rate (FPR)*, the probability of identifying a reference block *different* from what brute-force search finds. For FN cases, SFSketch compresses the data block using the LZ4 algorithm [15] because there is no reference block. For FP cases, SFSketch uses the Xdelta algorithm [56, 57] to perform delta compression of the block with the reference block that it identifies. For both cases, we measure the average data-reduction ratio (DRR) and compare it with that of brute-force search. Table 1 shows FNR, FPR, and DRR for FN/FP cases of the SFSketch-based reference search. DRR is normalized to that of brute-force search.

Table 1: Accuracy of LSH-based reference search [86].

Workload		PC	Install	Update	Synth	Sensor	Web	Avg.
FNR		35.3%	51.8%	56.3%	75.5%	48.1%	5.5%	35.7%
FPR		21.1%	15.8%	11.3%	14.1%	47.3%	60.6%	23.1%
DRR	FN cases	0.474	0.488	0.578	0.639	0.567	0.539	0.562
	FP cases	0.621	0.608	0.644	0.683	0.798	0.674	0.669

We make three observations from Table 1. First, the existing SFSketch-based technique suffers from high FNR (up to 75.5% and 35.7% on average), failing to find *any* reference block for many incoming blocks that actually have one or more reference blocks. Except for Web, SFSketch's FNR is higher than 35% for every workload. For FN cases, each data block is compressed by the LZ4 algorithm, and thus its DRR is considerably lower compared to when the block is delta-compressed with the reference block identified by brute-force search. As shown in Table 1, the normalized DRR in FN cases is 0.562 on average, showing that SFSketch provides 43.8% lower data reduction compared to the optimal for the FN cases (*i.e.*, for 35.7% of the entire data blocks on average).

Second, the SFSketch-based reference search frequently chooses a sub-optimal reference in some workloads, *e.g.*, Sensor and Web, which have a FPR of 47.3% and 60.6%, respectively. The sub-optimal selection of reference blocks results in lower data-reduction ratios over brute-force search. As shown in the last row in Table 1, the normalized DRR in FP cases is 0.669 on average, which means that SFSketch provides 33.1% lower data reduction compared to the optimal for the FP cases (*i.e.*, for 23.1% of the entire data blocks).

Third, FN cases are more common and have more negative impact on the DRR than FP cases. On average, FN cases occur for 35.7% of the incoming blocks, whereas FP cases occur for 23.1%. When a FN case happens, the data-reduction ratio using LZ4 is lower than when an FP case happens, which still uses delta compression albeit with a sub-optimal reference block; on average, the normalized DRR in FP cases (0.669) is 19% higher than that in FN cases (0.562).

The limited accuracy of SFSketch mainly stems from its inherent property; SFSketch is highly optimized to identify *only very similar* data. Considering the SF-based sketching process explained in Figure 2, it is highly unlikely that two blocks have at least one matching SF, unless they are very

---

[2]While brute-force search guarantees the highest data-reduction ratio for a workload, it is infeasible to use due to its prohibitively high performance overhead. For example, in our evaluation environments (see Section 5.1 for more detail), brute-force search takes more than 300 hours for the Install trace that writes a total of 8.83-GB data to the storage system.

similar. This property enables SFSketch to provide a high data-reduction ratio even when it selects a sub-optimal reference block for an incoming block (*i.e.,* for FP cases). However, it also causes SFSketch to frequently fail to find a *sufficiently good* reference block that is not very similar to the incoming block but is still beneficial for improving the data-reduction ratio.

It is challenging to optimize existing SF-based sketch algorithms to increase both FPR and FNR at the same time. The accuracy of SFSketch highly depends on its settings such as the number of features (m), the number of super features (N), the sliding window size (w), and the matching criteria. For example, under a matching criterion where two blocks are considered similar if they have at least one common SF, increasing the number of SFs (*i.e., N*) for each data block would reduce overall FNR. However, at the same time, it might increase FPR and reduce data-reduction ratios in FP cases because more dissimilar blocks could be chosen as reference blocks. Moreover, as shown in Table 1, the FNR/FPR trend of SFSketch-based search greatly varies across workloads, which makes it even more difficult to find the best configuration on average as well as on a per-workload basis. Instead, we investigate applicability of deep-learning algorithms for data sketching in delta compression, which can reduce the human effort for developing a new sketching scheme or fine-tuning existing techniques for different workloads.

3.2 Learning-to-Hash Method

The learning-to-hash method [43, 80] is a promising machine learning (ML)-based approach for the nearest-neighbor search problem. It trains a neural network (NN) to generate a hash value for a given input data block such that any two similar data blocks have similar hash values. Many prior works [10, 11, 23, 47, 48, 50, 89] demonstrate the high effectiveness of the learning-to-hash method at nearest-neighbor search in various applications, such as image recognition and classification.

Figure 3 depicts how a representative learning-to-hash scheme [50] generates a binary hash value of an image for content-based image retrieval. It extracts the hash value of an input image from the last hidden layer (*e.g.,* HL_N in Figure 3) of a NN that is trained to classify the input image to one of C possible classes. During inference, the activations in the last hidden layer of two similar images are likely to be largely the same if the two images belong to the same class. Therefore, their hash values should also be similar because they are directly extracted from the last layer by translating the output of each activation into a binary ('1' or '0').

The learning-to-hash method has potential to be used for

Generate a binary hash h(X)
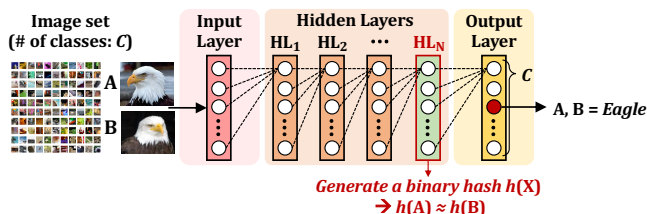➔ h(A) ≈ h(B)

Figure 3: Learning-to-hash for image retrieval.

reference block search in post-deduplication delta compression, another nearest-neighbor search problem. In particular, rapid advances in machine learning have enabled learning-based algorithms to outperform a human or human-made heuristics in various problems, such as facial recognition [65], speech recognition [83, 84], image classification [50], and system optimizations (*e.g.,* branch prediction [37, 38], memory scheduling [35], and prefetching [4]). These successful examples motivate us to develop a learning-based sketching scheme that could be more effective than existing LSH-based sketching schemes relying on human-designed heuristics and metrics.

4 DeepSketch

The key idea of DeepSketch is to use the learning-to-hash method to generate similar data signatures (*i.e.,* sketches) for any two data blocks that would provide a high data-reduction ratio when delta-compressed relative to each other. The main difference of DeepSketch over the existing post-deduplication delta-compression approach [75, 86] (described in Figure 1) is that DeepSketch generates a data block's sketch by using a deep neural network (DNN) model, instead of using an LSH function (*e.g., sk*(X) in Figure 1). For each incoming data block, DeepSketch performs DNN inference with the block as input and uses the resulting activation values in the DNN's last hidden layer as the block's sketch.

We envision that DeepSketch's DNN is *pre-trained* offline in other machines with more powerful computation resources before building or updating a storage machine. For example, to adopt DeepSketch in a new storage server of a data center, one can train DeepSketch's DNN using randomly-selected data blocks from existing storage servers that contain specific types of data (*e.g.,* databases, images, web caching, etc.) expected to be stored in the new storage server. Similarly, to further enhance the accuracy of DeepSketch, one can retrain DeepSketch's DNN and use the enhanced DNN to build new storage servers or reorganize existing ones.

While the high-level idea may sound simple, applying the learning-to-hash method for reference search in post-deduplication delta compression is not straightforward. This is because DeepSketch needs to deal with *general binary data*, which introduces the following two technical challenges:
Challenge 1. Lack of Semantic Information. The target data set of DeepSketch can contain *any* data from various applications, such as text, images, binary executable files, and so on. Compared to existing learning-to-hash approaches focusing on pre-categorized data (*e.g.,* Imagenet [20], CIFAR [42], and MNIST [44]), DeepSketch needs to process a much wider range of data without any well-defined semantic information about the delta-compressibility of data blocks.
Challenge 2. High Dimensional Space. The lack of semantic information in DeepSketch's target data sets leads us to perform *unsupervised learning* that is used for drawing inferences from a data set without labeled information. The most common unsupervised learning approach is to use a clustering algorithm that groups the target data set according to a certain

measure, *e.g.,* similarity of bit patterns in our case. However, possible bit patterns of a data block for DeepSketch have extremely high dimensional space (*e.g.,* $2^{4,096 \times 8}$ assuming a 4-KiB data block), which makes it difficult to 1) set a proper number of final clusters and 2) collect a data set large enough to cover all possible data patterns for a clustering algorithm.

To address the above challenges, we develop a new clustering algorithm, called *dynamic k-means clustering (DK-Clustering)*, which groups data blocks that would provide high delta-compression ratios when delta-compressed relative to each other (Section 4.1). To cope with potential groups of data blocks that are missing in the collected data sets, after clustering, we generate new data blocks by randomly and slightly modifying existing blocks. We then generalize the understanding of the similarity relationship between data blocks using the learning-to-hash method, so that DeepSketch can generate a learning-based data sketch for any given block (Section 4.2). With the sketch values computed by the learning-to-hash model, DeepSketch identifies the most similar reference block to each incoming block based on an approximate nearest-neighbor search technique (Section 4.3). We also perform hyper-parameter exploration for our DNN model to find the appropriate sketch size (Section 4.4).

4.1 Dynamic K-Means Clustering

DK-Clustering is based on the existing k-means clustering algorithm [53] that partitions a data set into a given number (*i.e., k*) of clusters such that each data element belongs to the cluster with the nearest mean value. Unfortunately, in our case, the value of k is initially unknown. Figuring out the most suitable value for k by exploring a given data set is time-consuming, considering the extremely high dimensionality of the data set that DeepSketch deals with.

The hierarchical clustering algorithm [39] is known to be suitable for such data sets, but it introduces prohibitive computation and memory overheads for a large-size data set. To be specific, the computation and space complexities of hierarchical clustering are $O(N^3)$ and $O(N^2)$, respectively, where N is the number of data blocks to cluster. This means that, for example, hierarchical clustering of a 4-GB data set requires TB-scale memory space assuming a data block size of 4 KiB.

There exist a number of *adaptive* clustering algorithms (*e.g.,* [5, 28, 63, 72, 87]) that aim to cluster a data set with limited knowledge of the number of final clusters. Unfortunately, using them for DNN training in DeepSketch is not straightforward either, because their efficiency also highly depends on the initial parameters that are set either randomly or manually, such as the initial number of clusters [28, 63, 72, 87] or the distance threshold to determine the similarity of given two objects [5]. Since the target data set of DeepSketch has an extremely high dimensional space while there is no available hint for good initial parameters, using existing techniques could either require significant effort to find appropriate initial parameters or lead to limited accuracy and/or prohibitive performance overhead due to the use of inappropriate initial parameters.

To overcome the above challenges, we develop DK-Clustering by extending the existing k-means clustering algorithm with specialized initialization steps to dynamically refine the value of k while clustering data without any hints for initial parameters. Figure 4 describes the overall process of DK-Clustering composed of two steps that are performed iteratively: **Step 1.** coarse-grained clustering and **Step 2.** fine-grained clustering. Coarse-grained clustering first creates rough clusters within an unlabeled data set, and then fine-grained clustering adjusts the assignment of data blocks by running a modified k-means clustering algorithm. Fine-grained clustering returns a data block to be unlabeled if the block is an *outlier* in the cluster, so that coarse-grained clustering can re-categorize the block at the next iteration. After Steps 1 and 2 converge, DK-Clustering repeats the above steps for *each* cluster in a recursive manner, which enables us to form fine-grained clusters that only contain data blocks sufficiently similar to each other.
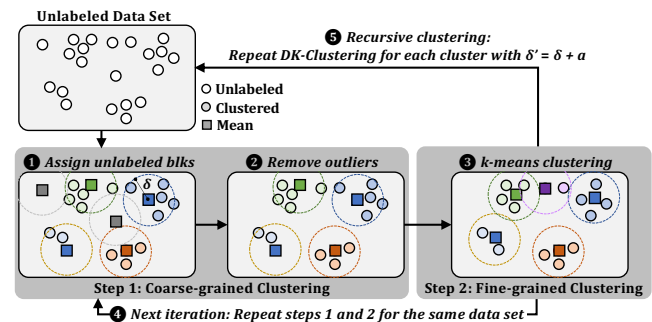

Figure 4: Overall procedure of dynamic k-means clustering.

Step 1: Coarse-Grained Clustering. Coarse-grained clustering takes a set of unlabeled blocks and clusters as the input, and aims to categorize all the unlabeled blocks. Initially, there exist only unlabeled blocks but no cluster, so DK-Clustering creates a new cluster and assigns the first block as the representative block (*i.e., mean*) of the cluster. After that, for each unlabeled block, DK-Clustering measures the data-reduction ratio when the block is delta-compressed with the mean of each cluster through the target delta-compression algorithm (*e.g.,* Xdelta [56,57]). DK-Clustering selects the cluster whose mean provides the highest data-reduction ratio for the unlabeled data block. If the data-reduction ratio is higher than a threshold δ, DK-Clustering adds the unlabeled block to the selected cluster. Otherwise, it creates a new cluster, and the unlabeled block becomes the new cluster's mean (❶ in Figure 4). After categorizing all unlabeled blocks, coarse-grained clustering removes clusters that contain only a single data block from the data set as there are likely no other blocks sufficiently similar to that block (❷).

Step 2: Fine-Grained Clustering. Since coarse-grained clustering roughly assigns unlabeled blocks to clusters, it cannot guarantee that all the data blocks belonging to the same cluster are sufficiently similar to each other. To address this, DK-Clustering performs fine-grained clustering for the resulting clusters from coarse-grained clustering. Fine-grained clustering performs a variant of k-means clustering, adjusting the

mean of each cluster and re-assigning each data block to the cluster containing the nearest mean (❸). Fine-grained clustering operates differently from the typical k-means clustering in three aspects. First, instead of Euclidean distance [19], it uses the delta-compression ratio of two data blocks as the distance function. Second, it derives a cluster's mean by *selecting* the block that provides the highest average data-reduction ratio when delta-compressed relative to each of the other blocks in the cluster. Third, if there is a data block whose delta-compression ratio when delta-compressed relative to the cluster's mean is lower than the threshold δ, DK-Clustering excludes the block from the cluster and considers it as an unlabeled block. After finishing fine-grained clustering on all the clusters, DK-Clustering repeats Steps 1 and 2 over all the clusters until no unlabeled data blocks exist (❹).

Step 3: Recursive Clustering. Fine-grained clustering guarantees that *every* resulting data block belongs to an appropriate cluster where the data block provides a data-reduction ratio higher than the given threshold δ when delta-compressed relative to the cluster's mean. Even though a sufficiently high value for δ would allow DK-Clustering to group only similar data blocks into the same cluster, other values for δ can provide better clustering results. In order to automatically find the best δ for a data set, once DK-Clustering reaches the convergence with a given threshold δ, it performs Steps 1 and 2 for *each* cluster using a new threshold $\delta' = \delta + \alpha$ in a recursive manner (❺). Data blocks assigned to each cluster are considered unlabelled again for the next recursion with the new threshold δ'. The recursion terminates when splitting a cluster shows no more benefit in improving the data-reduction ratio. More specifically, DK-Clustering stops the recursion for a cluster if the average data-reduction ratio of data blocks in the cluster is similar or lower than the average ratio of sub-clusters spawned from the cluster.

DK-Clustering Complexity. The space complexity of DK-Clustering is $O(N)$ since it only requires storing per-block information about which cluster the block belongs to. The computation complexity of DK-Clustering is $O(N \times K_F) + O(N^2/K_C) < O(N^2)$, where K_C and K_F are the number of total clusters after coarse-grained and fine-grained clustering steps, respectively. Although the number of iterations for DK-Clustering can vary depending on workload, DK-Clustering finishes within up to eight iterations for our training data sets. Note that, even for an extreme case where DK-Clustering requires a large number of iterations, we can easily limit the maximum number of iterations at minimal degradation in clustering quality. For example, one can set a threshold distance to finish DK-Clustering once it groups all data blocks such that any data block's distance from the corresponding cluster's mean is lower than the threshold distance.

4.2 Neural-Network Training

Figure 5 shows our method to train a DNN model for DeepSketch to generate a data block's sketch, which consists of two steps. In the first step, we train a *classification* model (❶) using the C_{TRN} clusters formed by DK-Clustering as differ-
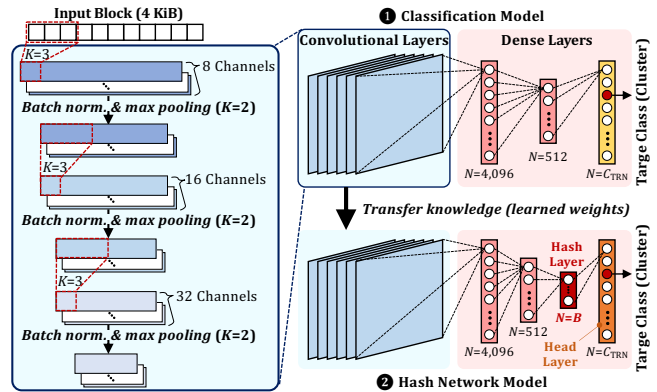

Figure 5: NN models of DeepSketch.

ent target classes. The first part consists of three standard 1D convolutional layers applying the max pooling and batch normalization techniques, which capture spatial locality of neighbor bytes within the data block. The network is then connected to dense layers to learn the relationship between the extracted spatial features and the target class.[3]

After training the classification model, in the second step, we transfer the learned knowledge of the classification model to a *hash* network model (❷). We employ a state-of-the-art learning-to-hash technique called GreedyHash [79]. We first initialize the hash network with the weights of the classification model. Instead of using the last layer of the classification model, we train the hash network with two different layers, a hash layer and a head layer, each of which learns the binary hash and class likelihood, respectively.

A key challenge in NN training for DeepSketch is that data blocks are not uniformly distributed over C_{TRN} clusters. In our data set, the largest 10% clusters contain 47.93% of the total data blocks. It would render training of the NN to be significantly biased towards specific bit patterns. To address this, we resize each of C_{TRN} clusters to have the same number of N_{BLK} blocks by 1) randomly selecting N_{BLK} blocks within a cluster containing more blocks than other clusters and 2) adding data blocks randomly and slightly modified from ones in a cluster containing fewer blocks.

Once training the hash network, the hash layer yields the B-bit representation for an input block, *i.e.,* the input block's sketch, allowing any two similar data blocks to have similar sketches with low Hamming distance. Note that, even if two data blocks do not belong to any of C_{TRN} clusters, we can infer their binary hash values based on the likelihood that each block belongs to the clusters, which dramatically improves the adaptability of our NN model over various data sets.

---

[3]We explore multiple NN structures and choose the one that provides the best classification accuracy and data-reduction ratio (shown in Figure 5). For example, when using a much simpler multi-layer perceptron (MLP) networks [24], DeepSketch hardly provides data-reduction benefits (less than 1%) over existing SF-based techniques. Adding the number of dense layers in the classification model in Figure 5 does not improve classification quality, either. We discuss detailed results for hyper-parameter search in Section 4.4.

4.3 Reference Selection

DeepSketch identifies whether or not any two given data blocks are similar by comparing the two blocks' sketches generated from the hash network model. A key challenge here is that the traditional exact-matching-based search method (which uses a hash table for the SK store) is not effective for the learning-to-hash model. For example, the hash network model may generate similar but few-bit different sketches for some blocks beneficial to be delta-compressed, which causes an exact-matching-based search method to misjudge those blocks to be dissimilar.

To address this issue, we use the approximate nearest-neighbor search (ANN) technique. Unlike the standard exact nearest-neighbor search, ANN techniques provide a scalable and performance-efficient way to find the most similar values by relaxing search conditions. In particular, we use the NGT library [16] that supports searching with high-dimensional binary data using neighborhood graphs and tree indexing.

Figure 6 illustrates the reference selection procedure of DeepSketch. For each incoming block, DeepSketch first computes its sketch, $\mathbf{H}$, using the hash network model. It then searches for the similar block from two SK stores. The first SK store utilizes the ANN technique, and DeepSketch queries it with $\mathbf{H}$ to get the data block with the most similar sketch, $\widehat{\mathbf{H}}$, in the ANN model. The other SK store buffers the sketches of R most-recently-written blocks. Let $\Delta(\mathbf{H}, \widehat{\mathbf{H}})$ be the Hamming distance between the two hash values. For each recent block in the buffer store, DeepSketch checks if there is a block with a Hamming distance smaller than $\Delta(\mathbf{H}, \widehat{\mathbf{H}})$. If there exists, DeepSketch chooses the block from the buffer store as the reference for the incoming block. Otherwise, it uses the block from the ANN-based SK store (*i.e.,* the block whose sketch is $\widehat{\mathbf{H}}$) as the reference.
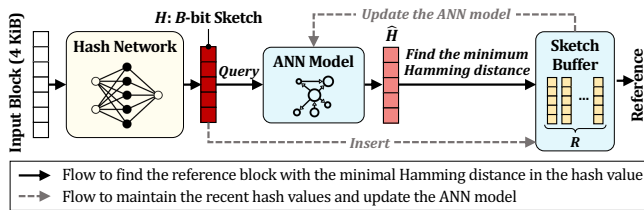

Figure 6: Overview of the reference selection procedure.

The underlying reason for using the two SK stores is that, under the current implementation using the NGT library, updating the ANN model takes a non-negligible amount of time. To avoid frequent updates of the data structure that would hurt the performance of DeepSketch, we design DeepSketch to update the ANN model in a batch by buffering the sketches of recently-written data blocks. When the number of sketches in the buffer exceeds a threshold T_{BLK} (*e.g.,* 128 in our default settings), DeepSketch flushes the buffered sketches to the ANN-based SK store. Note that it is important to check the sketch buffer in order to maximize the data-reduction ratio of DeepSketch. In our evaluation, 13.8% of the reference blocks are found in the sketch buffer on average (up to 33.8%).

4.4 Hyper-Parameter Exploration

This section presents our hyper-parameter exploration for DeepSketch to achieve high accuracy in reference search with a convolutional hash network.

Classification Model. As discussed in Section 4.2, the DNN training procedure of DeepSketch has two steps to train the classification model and the hash network model, respectively. To generate accurate sketches of data blocks, the classification model should predict the correct target classes (*i.e.,* the clusters formed by DK-Clustering). We identify the best hyper-parameters for the proposed classification model using the standard machine learning practice of the grid search along with nested cross-validation. We choose the number of the convolutional and dense layers from the grid $\langle 1, 2, 3 \rangle$, the number of the convolution channels size from $\langle 8, 16, 32, 64 \rangle$, the number of neurons for each dense layer from $\langle 512, 1,024, 2,048, 4,096 \rangle$, the dropout rate for the dense layers from $\langle 0.0, 0.1, 0.2, 0.5 \rangle$, and the learning rate from $\langle 0.01, 0.02, 0.005, 0.1, 0.5 \rangle$. We utilize ReLU for the activation function for each layer and train the model with the Adam optimizer [41]. We use 10% of samples in our data sets for training and the remaining 90% for testing. Finally, we select the proposed classification model structure that shows the best testing accuracy in the cross-validation.

Figure 7 shows the loss and testing accuracy changes over training epochs for the classification model. The proposed classification model accurately predicts the target cluster identified in DK-Clustering even though the data sets used in our evaluation has a relatively large number of the clusters, $C_{\mathrm{TRN}} = 34,025$. After training with 350 epochs, the model training procedure sufficiently converges, achieving 93.42% for Top-1 and 96.02% for Top-5 accuracy. It implies that the deep learning method itself can accurately identify similar blocks for an incoming block without any other information.
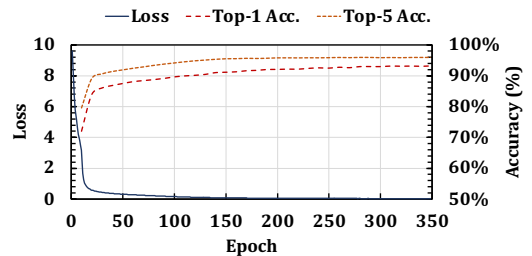

Figure 7: Loss and accuracy of classification model.

Hash Network Model. Next, we train the hash network model while changing the sketch size B. With the smaller B, similar data blocks would have a higher chance to have the same hash value, but it also increases the false-positive rate, *i.e.,* dissimilar blocks belonging to different clusters would have the same hash value. One may set the number of bits with a sufficiently large number, but doing so increases the memory overhead for the SK store and the computation time for ANN search and update processes.

To determine the best sketch size, we verify when the hash network model could achieve the classification model's original accuracy. Recall that the hash network model learns both

the hash coding and classification at the same time. Thus, we can verify whether it correctly classifies the target class by checking the last head layer's activations. Figure 8 summarizes our evaluation results. We evaluate three candidate values for B, 32, 64, and 128, over different learning rates λ. Note that the model does not converge when $B = 128$ and $\lambda = 0.005$, so we omit the results. The results show that the hash network model does not recover the accuracy of the classification model with the small hash bits, 32 and 64, since the representation capability of the hash coding is insufficient. When $B = 128$, we observe that the hash network model also predicts the target clusters with a high accuracy, *e.g.,* it achieves the Top-5 accuracy of 96.92% with $\lambda = 0.002$, exceeding the original target accuracy of the classification model. Thus, we decide to use $B = 128$ for our implementation of DeepSketch.
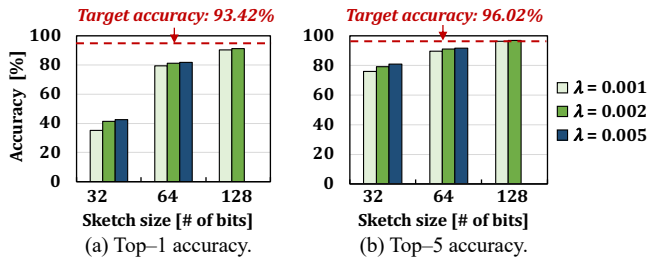

Figure 8: Accuracy of hash network model.

5 Evaluation

In this section, we evaluate the data-reduction benefits and performance/memory overheads compared to the state-of-the-art super feature (SF)-based sketching technique [86].

5.1 Methodology

Evaluation Platform. We develop a post-deduplication delta-compression platform that is used as a general workbench to implement and evaluate various reference search techniques.[4] Our platform runs on a server machine that employs Intel's Xeon 4110 CPU with 8 cores running at 2.1 GHz, 128-GB DDR4 DRAM, and 8 Samsung 860PRO 1-TB SSDs, while using GeForce RTX 2080 for DNN inference in DeepSketch.

Our platform operates as described in Figure 1; for every host write, it performs deduplication, delta compression, and lossless compression in order. It maintains three main data structures: 1) a fingerprint store for deduplication, 2) a sketch store for delta compression, and 3) a reference table for serving future read requests. The data block size is 4 KiB, which is identical to the default block size of widely-used file systems [18, 60]. We use the MD5 cryptographic hash algorithm [69] to generate a 128-bit fingerprint of an incoming data block and the Xdelta delta-compression algorithm [56, 57] to compress a non-deduplicated data block with its reference block. If there are multiple reference blocks similar to an incoming data block, our platform uses the first-found

---

one as a reference by default. When the platform cannot find a reference block, it compresses the incoming block using the LZ4 algorithm [15]. We set the threshold for the number of buffered sketches to invoke ANN updates to 128, which we empirically determine to minimize the performance overhead of exhaustive search and prevent too frequent ANN updates.

Baseline Technique. We compare DeepSketch against *Finesse* [86], the state-of-the-art SF-based technique that provides much higher throughput while retaining almost the same data-reduction ratio compared to the representative post-deduplication delta-compression technique [75]. We configure Finesse using the default settings presented in [86], which are already optimized and have been shown to provide the best data-reduction efficiency with low overhead for a wide range of workloads. Finesse generates three 192-bit SFs, each of which can be obtained by transposing four features from different hash functions (*i.e.,* twelve (= 3×4) Rabin fingerprint functions [68] with a window size of 48 bytes are used in total). It considers that two data blocks are similar if they have one or more matching SFs, and selects the data block that has the largest number of matching SFs with the incoming block as the reference block for delta compression.

Workloads. We use eleven block I/O traces that we collect by running different applications on real systems and capturing write requests (including the requested data) to the storage devices. There is no backup process during trace collection. Table 2 summarizes the characteristics of the traces in terms of the size, deduplication ratio (*i.e., Orignial Data-Set Size / Data-Set Size after Deduplication*), and average compression ratio (*i.e., Original Data-Set Size / Compressed Data-Set Size*). We collect the I/O traces including contents written in the storage system from real desktop machines and servers while running different applications.

Table 2: Summary of the evaluated workloads.

Workload	Description	Size	Dedup. ratio	Comp. ratio
PC	General Ubuntu PC usage	1.57 GB	1.381	2.209
Install	Installing & executing programs	8.83 GB	1.309	2.45
Update	Updating & downloading SW packages	3.73 GB	1.249	2.116
Synth	Synthesizing hardware modules	653 MB	1.898	2.083
Sensor	Sensor data in semiconductor fabrication	91.2 MB	1.269	12.38
Web	Web page caching	959 MB	1.9	6.84
SOF0		8.98 GB	1.007	2.088
SOF1	Storing Stack Overflow database [33]	13.6 GB	1.01	1.997
SOF2	as of 2010 (SOF0) and 2013 (SOF1–4)	13.6 GB	1.01	1.996
SOF3		13.6 GB	1.01	1.997
SOF4		13.6 GB	1.01	1.996

For DeepSketch, we use different sets of data for training and testing. In order to evaluate the *adaptability* of DeepSketch (*i.e.,* how well DeepSketch operates under a workload totally different from ones used in its DNN training), we do *not* use the five traces collected from Stack Overflow database [33] (SOF0–4) for training the DNN of DeepSketch. By default, we train DeepSketch's DNN model using a single data set that contains 10% of all the remaining six traces and evaluate DeepSketch with the remaining 90% of the six traces and entire SOF traces.

5.2 Overall Data Reduction

Figure 9 shows the data-reduction ratio after post-deduplication delta compression with the two reference search techniques, Finesse and DeepSketch, under the eleven workloads. We only present SOF1 as a representative result of SOF1-4 as they show little variations lower than 0.01%. All values are normalized to the data-reduction ratios of a baseline system that performs only deduplication and lossless compression in order, which we call *no delta compression (noDC)*.
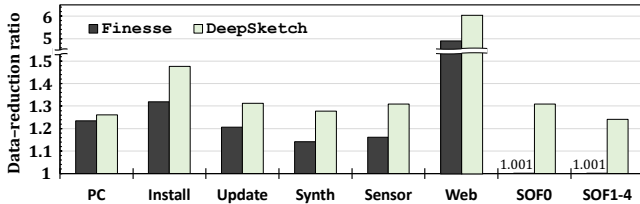

Figure 9: Comparison of overall data-reduction ratio.

We make two main observations from Figure 9. First, DeepSketch significantly outperforms Finesse in most workloads. Except for PC in which DeepSketch provides the similar data-reduction ratio with Finesse, DeepSketch exhibits up to 33% (on average 21%) higher data-reduction ratios than Finesse. In particular, DeepSketch greatly improves the data-reduction ratio by *at least* 24% over Finesse under SOF workloads. This suggests that 1) DeepSketch can improve the data-reduction efficiency for workloads that the state-of-the-art SF-based search technique cannot effectively cope with, and 2) DeepSketch has high adaptability (*i.e.,* it can work efficiently for data sets that are not used for the DNN training). Second, DeepSketch provides higher data-reduction ratios even for highly compressible workloads. Under Web, Finesse significantly reduces the write traffic by about 80% over noDC, but DeepSketch increases the data-reduction ratio even further by 33% compared to Finesse. From our observations, we conclude that DeepSketch is an effective solution to maximize the data-reduction ratio for various workloads.

5.3 Reference Search Pattern Analysis

To better understand how DeepSketch can outperform the state-of-the-art technique, we analyze the reference-search efficiency of DeepSketch and Finesse. Given a data block B, we measure $S_{FS}(B)$ and $S_{DS}(B)$, the number of *saved bytes* by Finesse and DeepSketch, respectively. $S_{FS}(B)$ (or $S_{DS}(B)$) is obtained by subtracting the size of B when delta-compressed with the reference block found by Finesse (or DeepSketch) from the original size of B (*i.e.,* 4 KiB). The larger the $S_{FS}(B)$ (or $S_{DS}(B)$) value, the higher the reference-search efficiency. If a reference search technique fails to find a reference block for B, we compress it using the LZ4 algorithm and then use the compressed size to calculate data saving.

Figure 10 plots coordinates of $x = S_{FS}(B_i)$ and $y = S_{DS}(B_i)$ for a block B_i in each workload. If $x = y$, Finesse and DeepSketch exhibit the same delta-compression ratio (highlighted with a red line in Figure 10), which implies that they select the same reference block. A coordinate $(S_{FS}(B_i), S_{DS}(B_i))$ above
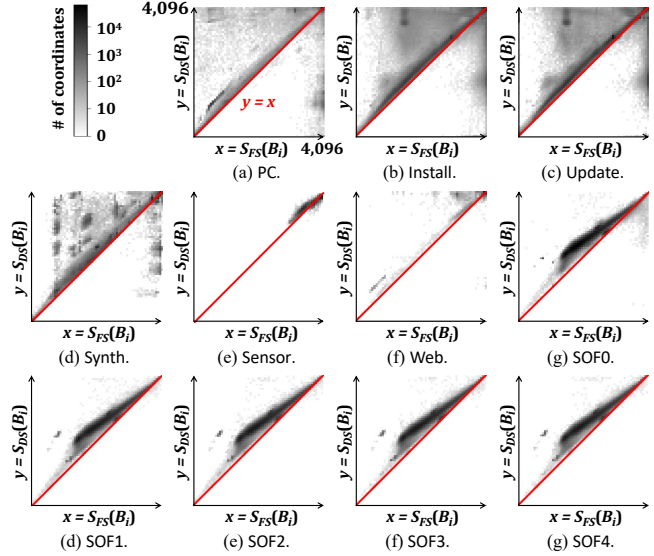

Figure 10: Comparison of the reference-search pattern.

(or below) the line means that DeepSketch provides higher (or lower) data-reduction ratio for block B_i than Finesse.

From Figure 10, we make three observations. First, as expected, DeepSketch provides higher data savings compared to Finesse for a large number of blocks under every workload. Second, despite the higher data savings of DeepSketch over Finesse in general, there are also a non-trivial number of blocks for which Finesse selects better references, achieving higher data savings than DeepSketch. Excluding the SOF workloads, Finesse selects higher-quality references compared to DeepSketch for up to 11.8% of the total blocks. Third, DeepSketch and Finesse show quite different patterns in reference search. As shown in Figure 10, the coordinates in $y > x$ region (*i.e.,* where DeepSketch outperforms Finesse) are close to the line $y = x$, and at the same time, many of them are scattered across a wide range of the region compared to the coordinates in $y < x$ region. On the other hand, a majority of the coordinates in $y < x$ region (*i.e.,* where Finesse outperforms DeepSketch) tend to have a very large y value (*e.g.,* > 3,072). These imply that, while Finesse is effective to find a reference highly similar to an input block, it also misses a number of blocks that DeepSketch can find and use to improve the data-reduction efficiency.

5.4 Combination with Existing Techniques

Our second and third observations in Section 5.3 motivate us to combine DeepSketch with existing techniques to maximize the data-reduction ratio. We design a storage system that employs both Finesse and DeepSketch. When the two techniques find different reference blocks for an incoming block, the system chooses the one that provides a higher data-reduction ratio. Such an approach increases the memory and computation overheads for data sketching but would be desirable for a system where data reduction is paramount (*e.g.,* backup systems). We leave the study of efficiently combining DeepSketch with existing techniques as future work.

Figure 11 shows the combined approach's data-reduction benefits compared to when using either Finesse or DeepSketch alone.[5] We also measure the *optimal* data-reduction ratio (*i.e.,* when every data block is delta-compressed with the best reference block found by brute-force search) for each workload to understand room for improvement after applying the combined approach. To emphasize the benefits of the combined approach over the standalone techniques, we normalize all the results in Figure 11 to Finesse.
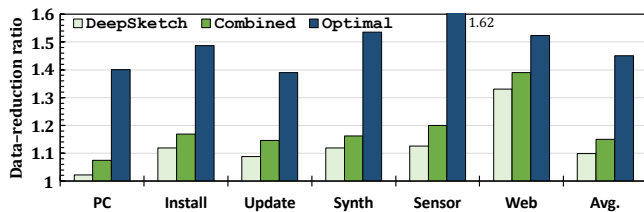
Figure 11: Data-reduction improvement of a combination of DeepSketch and Finesse.

We observe that, as expected, the combined approach further improves the data-reduction ratio compared to the two standalone techniques under most workloads. The combined approach achieves up to 38% and 6.6% (15% and 4.8% on average) data-reduction improvements over Finesse and DeepSketch, respectively. We also observe that the combined approach can reduce the gap in data-reduction ratios between the existing reference search techniques and the optimal. Although there is still large room for improvement (*i.e.,* up to 35% and 26% on average) even after applying the combined approach, the combined approach reduces the gap by up to 81% (*i.e.,* 62% → 9.6% under Web) and by 42% on average. From our observations, we conclude that DeepSketch can also be used as a useful method to complement the weakness of existing post-deduplication delta-compression techniques.

5.5 Impact of Training Data-Set Quality

We evaluate the impact of the training data-set quality on the data-reduction ratio of DeepSketch. Figure 12 shows the average data-reduction ratio of DeepSketch for the entire workloads listed in Table 2 when we use two different types of training data sets. First, we evaluate how DeepSketch's benefit changes when we train its DNN model using 1%/2%/3%/5%/10% of the *entire* data sets (the blue line in Figure 12). Second, we measure DeepSketch's benefit when we use 10% of requests only from Sensor for DNN training (the dashed red line in Figure 12). When we use $x\%$ ($< 10\%$) of each trace for training, we use the remaining $(100-x)\%$ to evaluate the data-reduction ratio of DeepSketch. All values in Figure 12 are normalized to the data-reduction ratio obtained when using 10% of the entire data sets for DNN training.

We make two observations from Figure 12. First, while a larger training data set increases DeepSketch's benefit, DeepSketch can provide a fairly good data-reduction ratio even with

[5]We omit the results of the SOF workloads in Figure 11 because there is no motivation to combine DeepSketch with Finesse under such workloads for which Finesse provides negligible data reduction.
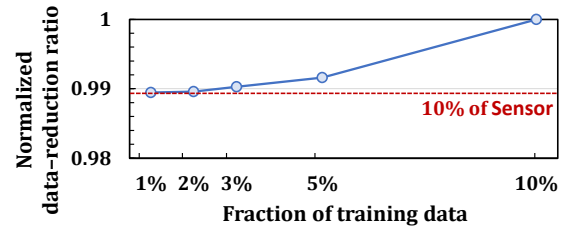

Figure 12: Effect of training data set on data-reduction ratio.

a very small training data set. Using only 1% of the traces for DNN training provides 98.9% of the data-reduction ratio obtained when using 10% of the traces. Second, DeepSketch can provide a high data-reduction ratio even when we use a training data set collected from a single trace. Compared to when we use 10% of all traces for DNN training, using 10% of only Sensor decreases the data-reduction ratio by less than 1%. Based on our observations, we conclude that it is possible to train an effective DNN model for DeepSketch with a limited data set, while providing high adaptability for diverse input data sets.

To study the detailed impact of the training data-set quality, we analyze how the sketches generated by DeepSketch change with different training data sets. To this end, we measure the average *data-saving ratio* (*i.e.,* $1-$ *Delta-Compressed Data Size* / *Original Data Size*) of delta-compressed blocks depending on the Hamming distance between the sketches of the input and reference blocks (*i.e.,* $\Delta(\mathbf{H},\widehat{\mathbf{H}})$ in Section 4.3.) Figure 13 shows the relationship between the data-saving ratio and sketch Hamming distance for three different DNN models trained with 10% of Sensor (10%-Sensor) and 1%/10% of all traces (1%-All and 10%-All). In general, the higher the data-saving ratio at a low sketch Hamming distance, the more accurate the sketches generated by DeepSketch.
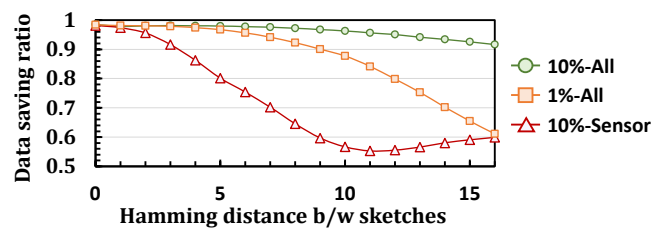

Figure 13: Effect of training data set on sketch accuracy.

We identify the following two findings from Figure 13. First, for all DNN models, DeepSketch provides extremely high data saving (close to 1) when $\Delta(\mathbf{H},\widehat{\mathbf{H}}) \le 2$. The result shows that all the three DNN models enable DeepSketch to identify highly similar data blocks by generating almost identical sketches. It is due to the nature of the DNN-based learning-to-hash method: a DNN can be easily trained to yield the same hash values for the data with negligible differences. Second, in 1%-All and 10%-Sensor, the data-reduction ratio degrades more significantly as the Hamming distance increases, compared to 10%-All. It suggests that we can further improve the benefit of DeepSketch by increasing the accuracy of the sketch generation with a better DNN model, *e.g.,* using high-quality data sets and/or advanced model ar-

chitectures. In the current version of DeepSketch, the ANN model compensates for such potential accuracy loss by finding sufficiently-good reference blocks with best efforts.

5.6 Overhead Analysis

Performance Overhead. Figure 14 shows the average throughput of DeepSketch and the combined approach of DeepSketch and Finesse under different workloads, normalized to Finesse.[6] DeepSketch and combined approach provide up to 73.7% and 44.9% (44.6% and 28.4% on average across all workloads) of the average throughput of Finesse. This non-trivial performance overhead is due to the inherent trade-off between the data-reduction ratio and throughput in post-deduplication delta compression; performing delta compression for more data blocks would increase the data-reduction ratio, but it comes at the cost of performance degradation since delta compression takes more time compared to lossless compression (*e.g.,* in our current implementation, LZ4 takes 6.9 μs per block on average, which is less than 10% of the average execution time of Xdelta).
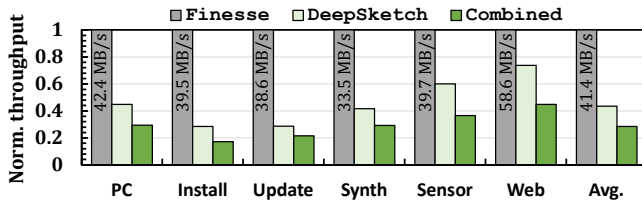

Figure 14: Performance overhead of DeepSketch.

To better understand the performance overheads of DeepSketch, we measure the average latency of each step per input data block during the post-deduplication delta-compression process. DeepSketch requires two modifications on existing techniques, 1) replacing the SF-based sketching engine with the DNN-based one and 2) using the ANN engine described in Section 4.3 as the SK store. For fair comparison, we implement the SK store of Finesse using the unordered-map data structure that provides $O(1)$ time complexity for lookup. Figure 15 visualizes the fraction of the average time spent for each step in the entire data-reduction process.
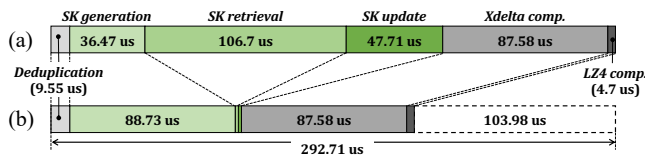

Figure 15: Average latency for each data-reduction step in (a) DeepSketch and (b) Finesse.

As shown in Figure 15, DeepSketch and Finesse operate differently in only three steps of the entire data-reduction process: 1) sketch generation for an incoming block, 2) sketch retrieval from the SK store, and 3) sketch update to the SK

store. The other steps, including deduplication, Xdelta compression, and LZ4 compression, are performed in the same ways. Due to the simplicity of the hash network model network and GPU acceleration, DeepSketch reduces the latency of sketch generation from 88.73 μs to 36.47 μs (by 58.9%) over Finesse. However, using the ANN engine significantly increases the latencies for sketch retrieval and update, leading to 55.1% increase in the total average latency over Finesse.

The performance overhead of DeepSketch over Finesse is non-trivial, but it would not be a serious obstacle for its wide adoption due to two reasons. First, we target a system where data reduction is critically important so that DeepSketch's benefits outweigh its performance overheads. Second, the performance overhead of DeepSketch can be mitigated in several ways. For example, if the data-reduction process is performed in *background*, its negative performance impact could be relatively small. We can also leverage the parallelism of multi-core CPUs to optimize software modules. For example, the sketch update procedure can be performed in parallel with other modules. This hides the cost of updating sketches during the compression steps, thereby reducing the performance overhead by 45.8% (*i.e.,* 103.98 μs $\rightarrow$ 56.27 μs).

Memory Overhead. Like existing post-deduplication delta-compression techniques [75, 86], DeepSketch inevitably requires additional memory space for the sketch store. Despite the smaller sketch size of DeepSketch compared to existing techniques [75, 86] (128 bits vs. 192 bits), DeepSketch's memory overhead might be unacceptable if it keeps track of the sketches of *all* non-deduplicated data blocks. For example, suppose that the data block size is 4 KiB, and 80% of the stored data is unique (*i.e.,* non-deduplicated). Then, the required memory space for the sketch store is about 0.3% ($0.8 \times 16/4,096$) of the size of the stored data (e.g., around 100-GB memory space to work with 32-TB data).

However, the memory overhead would not be a significant obstacle to use DeepSketch in practice for two reasons. First, the memory overhead for the sketch store is a common problem in all the sketch-based techniques. Second, prior works demonstrate that a small fraction of data blocks are frequently used as the reference block for many input blocks [26, 64]. Thus, keeping only most-frequently-used sketches in a limited-size sketch store (i.e., a with least-frequently-used (LFU) eviction policy) would provide sufficiently high compression efficiency. We leave such further optimizations to mitigate DeepSketch's memory overhead for future work.

6 Discussion

Scalability to Larger Data Sets. As shown in our evaluation, DeepSketch provides high data-reduction ratios even for workloads that are not used in DNN training (*e.g.,* SOF workloads), which implies the high generalizability of DeepSketch. Nevertheless, due to the limited amount of data sets publicly accessible, it is difficult to say whether DeepSketch would be effective under *any* given workload. For example, DeepSketch may require a larger DNN model to provide suf-

---

[6]Note that DNN training does *not* affect DeepSketch's throughput because it can be performed *offline* as explained in Section 4. In our system described in Section 5.1, DNN training (including DK-Clustering) takes less than 4 hours with 300 epochs for our 1.6-GB training data set.

ficient benefits for large data sets (that we do not observe in this work), which would significantly increase the training overheads of DeepSketch. However, we believe that DeepSketch would be able to work for larger data sets due to three reasons. First, DeepSketch 's DNN model has much smaller computation complexity than state-of-the-art DNN models, so there is significant room for DeepSketch to use the larger DNN models. Second, the memory space required for training depends more on the size of the DNN model rather than the size of the training data set. Our current model is only a few hundreds of megabytes in size, which can be run on a single commonly-used GPU. Third, as explained in Section 4, DNN training can be performed offline in different machines with more computing/memory resources.

Cost-Effectiveness of DeepSketch. The current version of DeepSketch requires a powerful GPU for DNN inference/training and thus introduces non-trivial performance and power overheads. However, we believe that such overheads would not be a significant obstacle for the wide adoption of DeepSketch due to two reasons. First, as explained in Section 4, DNN training can be done in different machines (*e.g.,* in cloud servers) without requiring frequent retraining, and multiple storage servers (that store similar types of data) can use the same DNN model, amortizing the training cost. Second, there has been significant effort to develop high-performance and energy-efficient accelerators for both light-weight DNN inference (e.g., [6, 27, 66, 73]) and ANN search (e.g., [32, 40, 71]), which would greatly reduce the performance, power, and resource overheads of DeepSketch.

7 Related Work

To our knowledge, this work is the first to propose a learning-based data-sketching technique for accurate reference search in storage-level delta compression. As we have already discussed state-of-the-art techniques closely related to DeepSketch in Sections 2 and 3, in this section, we briefly discuss other recent works on 1) storage-level data reduction and 2) machine learning-based video/image compression.

Storage-level Data Reduction. The fundamental ideas of data-reduction techniques were proposed several decades ago. Hence, their theoretical properties and limitations, in terms of data reduction, have been studied intensively. Recent studies focus more on how to efficiently deploy them to various platforms to achieve space savings with faster compression speeds, lower computation costs, and less energy consumption [12, 26, 46, 64, 85, 86]. For example, SmartDedup [85] proposes a low-cost deduplication technique for resource-constrained devices where computing resources as well as energy budget are seriously limited. Finesse [86] is a representative example of enhancing delta-compression speed without loss in data-reduction ratio by relaxing the complexity of sketch generation. Some prior works [12, 26, 46] present that deduplication and compression could be integrated in an SSD controller to improve storage lifetime as well as performance.

The key difference of this study from the above recent works is that this work presents a new direction to improve

data-reduction ratio, the fundamental goal of a data-reduction technique. Our work analyzes the limitations of probabilistic and statistical approaches and shows that emerging deep-learning methods can be promising alternatives to and/or complements the traditional methods.

Machine Learning for Video and Image Compression. Several works attempt to improve video/image compression efficiency using machine learning [2, 13, 14, 54, 61, 77, 77]. To enhance existing video compression algorithms, some leverage CNNs [13, 14, 77] and others employ Long Short-Term Memory (LSTM) networks to learn video representations [77] and predict future frames [54]. Their common idea is to accurately predict pixel values of next video frames and store only deltas for reconstruction. More recent studies use Generative Adversarial Networks (GAN) to generate part or all of the image content from a semantic label map [2, 61]. They achieve space savings by storing only a smaller amount of preserved data and the label map in storage devices.

DeepSketch is different from these studies in two aspects. First, unlike existing ML-based compression methods that target images and videos, DeepSketch aims to compress binary data, which requires handling extremely high-dimensional data sets without any semantic information. Second, ML-based compression methods are basically lossy compression algorithms, but our system is a lossless compression system that enables us to reconstruct original data without any data loss.

8 Conclusion

We introduce DeepSketch, the first learning-based reference search technique to improve the data-reduction efficiency of post-deduplication delta compression. DeepSketch uses the learning-to-hash method to overcome the limitations of existing techniques that miss a number of good reference candidates for delta compression of incoming data blocks. We present a new deep neural network training method that enables DeepSketch to efficiently learn delta-compression-aware data representation for unlabeled data sets with an extremely high dimensional space. Using various real-world data sets, we experimentally demonstrate that DeepSketch is an efficient solution not only as a replacement for but also as a complement to existing reference search techniques, significantly reducing the data-reduction gap from the optimal.

References

[1] DeepSketch GitHub repository, 2018. https://github.com/dgist-datalab/deepsketch-fast2022.

[2] Eirikur Agustsson, Michael Tschannen, Fabian Mentzer, Radu Timofte, and Luc Van Gool. Generative adversarial networks for extreme learned image compression. In *ICCV*, October 2019.

[3] Miklos Ajtai, Randal Burns, Ronald Fagin, Darrell D. E. Long, and Larry Stockmeyer. Compactly encoding unstructured inputs with differential compression. *JACM*, 2002.

[4] Rahul Bera, Konstantinos Kanellopoulos, Anant Nori, Taha Shahroodi, Sreenivas Subramoney, and Onur Mutlu. Pythia: A customizable hardware prefetching framework using online reinforcement learning. In *MICRO*, 2021.

[5] Sanjiv K. Bhatia. Adaptive k-means clustering. In *FLAIRS*, 2004.

[6] Amirali Boroumand, Saugata Ghose, Berkin Akin, Ravi Narayanaswami, Geraldo F. Oliveira, Xiaoyu Ma, Eric Shiu, and Onur Mutlu. Google neural network models for edge devices: Analyzing and mitigating machine learning inference bottlenecks. In *PACT*, 2021.

[7] Andrei Z. Broder, Moses Charikar, Alan M. Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *JCSS*, 2000.

[8] Randal Burns, Larry Stockmeyer, and Darrell D. E. Long. In-place reconstruction of version differences. *IEEE TKDE*, 2003.

[9] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. In *ASPLOS*, 1992.

[10] Yue Cao, Mingsheng Long, Bin Liu, and Jianmin Wang. Deep cauchy hashing for hamming space retrieval. In *CVPR*, 2018.

[11] Yue Cao, Mingsheng Long, and Jianmin Wang. Collective deep quantization for efficient cross-modal retrieval. In *AAAI*, 2017.

[12] Feng Chen, Tian Luo, and Xiaodong Zhang. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *USENIX FAST*, 2011.

[13] T. Chen, H. Liu, Q. Shen, T. Yue, X. Cao, and Z. Ma. DeepCoder: a deep neural network based video compression. In *VCIP*, 2017.

[14] Z. Chen, T. He, X. Jin, and F. Wu. Learning for video compression. *IEEE TCSVT*, 30(2), 2020.

[15] Yann Collet. LZ4 – extemely fast compression algorithm. http://lz4.github.io/lz4/.

[16] Yahoo! Japan Corp. Neighborhood graph and tree for indexing high-dimensional data. https://github.com/yahoojapan/NGT.

[17] Alvin Cox. JEDEC SSD endurance workloads. In *FMS*, 2011.

[18] Helen Custer. *Inside the Windows NT File System*. Microsoft Press, 1994.

[19] Per-Erik Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 1980.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.

[21] Wei Dong, Fred Douglis, Kai Li, R Hugo Patterson, Sazzala Reddy, and Philip Shilane. Tradeoffs in scalable data routing for deduplication clusters. In *USENIX FAST*, 2011.

[22] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: a scalable secondary storage. In *USENIX FAST*, 2009.

[23] Venice Erin Liong, Jiwen Lu, Gang Wang, Pierre Moulin, and Jie Zhou. Deep hashing for compact binary codes learning. In *CVPR*, 2015.

[24] Matt W. Gardner and Stephen R. Dorling. Artificial neural networks (the multilayer perceptron) – a review of applications in the atmospheric sciences. *Atmospheric Environment*, 1998.

[25] Jim Gray and Catharine Van Ingen. Empirical measurements of disk failure rates and error rates. *arXiv*, 2007.

[26] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing NAND flash-based SSDs. In *USENIX FAST*, 2011.

[27] Ramyad Hadidi, Jiashen Cao, Yilun Xie, Bahar Asgari, Tushar Krishna, and Hyesoon Kim. Characterizing the deployment of deep neural networks on commercial edge devices. In *IISWC*, 2019.

[28] Greg Hamerly and Charles Elkan. Learning the k in k-means. In *NeurIPS*, 2003.

[29] Geoffrey E. Hinton, Terrence Joseph Sejnowski, et al. *Unsupervised Learning: Foundations of Neural Computation*. MIT press, 1999.

[30] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Lass, and Keith Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. In *USENIX NSDI*, 2017.

[31] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proc. IRE*, 1952.

[32] Mohsen Imani, Yeseong Kim, and Tajana Rosing. Nngine: Ultra-efficient nearest neighbor accelerator based on in-memory computing. In *ICRC*, 2017.

[33] Stack Exchange Inc. Stack Exchange data dump. `https://archive.org/details/stackexchange`.

[34] Piotr Indyk, Rajeev Motwani, Prabhakar Raghavan, and Santosh Vempala. Locality-preserving hashing in multi-dimensional spaces. In *STOC*, 1997.

[35] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *ISCA*, 2008.

[36] Navendu Jain, Michael Dahlin, and Renu Tewari. Taper: Tiered approach for eliminating redundancy in replica synchronization. In *USENIX FAST*, 2005.

[37] Daniel A. Jiménez. Fast path-based neural branch prediction. In *MICRO*, 2003.

[38] Daniel A. Jiménez and Calvin Lin. Dynamic branch prediction with perceptrons. In *HPCA*, 2001.

[39] Stephen C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 1967.

[40] Himanshu Kaul, Mark A. Anders, Sanu K. Mathew, Gregory Chen, Sudhir K. Satpathy, Steven K. Hsu, Amit Agarwal, and Ram K. Krishnamurthy. 14.4A 21.5M-query-vectors/s 3.37 nJ/vector reconfigurable k-nearest-neighbor accelerator with adaptive precision in 14nm tri-gate CMOS. In *ISSCC*, 2016.

[41] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *ICLR*, 2015.

[42] Alex Krizhevsky. Learning multiple layers of features from tiny images, 2009.

[43] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *NeurIPS*, 2009.

[44] Yann LeCun. The MNIST database of handwritten digits, 1998.

[45] Sungjin Lee, Taejin Kim, Jisung Park, and Jihong Kim. An integrated approach for managing the lifetime of flash-based SSDs. In *DATE*, 2013.

[46] Sungjin Lee, Jihoon Park, Kermin Fleming, and Jihong Arvind, Kim. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE TCE*, 2011.

[47] Cong Leng, Jiaxiang Wu, Jian Cheng, Xi Zhang, and Hanqing Lu. Hashing for distributed data. In *ICML*, 2015.

[48] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *USENIX ATC*, pages 395–410, 2019.

[49] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX FAST*, 2009.

[50] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. Deep learning of binary hash codes for fast image retrieval. In *CVPR*, 2015.

[51] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In *USENIX FAST*, 2014.

[52] Weiqiang Liu, Faqiang Mei, Chenghua Wang, Maire O'Neill, and Earl E. Swartzlander. Data compression device based on modified LZ4 algorithm. *IEEE TCE*, 2018.

[53] Stuart Lloyd. Least squares quantization in PCM. *IEEE TIT*, 1982.

[54] William Lotter, Gabriel Kreiman, and David D. Cox. Deep predictive coding networks for video prediction and unsupervised learning. *CoRR*, abs/1605.08104, 2016.

[55] Jingwei Ma, Gang Wang, and Xiaoguang Liu. DedupeSwift: Object-oriented storage system based on data deduplication. In *TrustCom*, 2016.

[56] Josh MacDonald. Xdelta: Open-source binary diff, differential compression tools, VCDIFF (RFC 3284) delta compression. `http://xdelta.org`.

[57] Josh MacDonald. *File system support for delta compression*. PhD thesis, 2000.

[58] Thanos Makatos, Yannis Klonatos, Manolis Marazakis, Michail D. Flouris, and Angelos Bilas. Using transparent compression to improve SSD-based I/O caches. In *EuroSys*, 2010.

[59] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *USENIX FAST*, 2016.

[60] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium*, 2007.

[61] Fabian Mentzer, George Toderici, Michael Tschannen, and Eirikur Agustsson. High-fidelity generative image compression. In *NeurIPS*, 2020.

[62] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *USENIX FAST*, 2011.

[63] S. Ehsan Yasrebi Nayini, Somayeh Geravand, and Ali Maroosi. A novel threshold-based clustering method to solve k-means weaknesses. In *ICECDS*, 2017.

[64] Jisung Park, Sungjin Lee, and Jihong Kim. DAC: Dedup-assisted compression scheme for improving lifetime of NAND storage systems. In *DATE*, 2017.

[65] P. Jonathon Phillips, Amy N. Yates, Ying Hu, Carina A. Hahn, Eilidh Noyes, Kelsey Jackson, Jacqueline G. Cavazos, Géraldine Jeckeln, Rajeev Ranjan, Swami Sankaranarayanan, Jun-Cheng Chen, Carlos D. Castillo, Rama Chellappa, David White, and Alice J. O'Toole. Face recognition accuracy of forensic examiners, superrecognizers, and face recognition algorithms. *Proc. NAS*, 2018.

[66] Haotong Qin, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. Binary neural networks: A survey. *Pattern Recognition*, 2020.

[67] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *USENIX FAST*, 2002.

[68] Michael O. Rabin. Fingerprinting by random polynomials. *Technical Report*, 1981.

[69] Ronald Rivest and S. Dusse. The MD5 message-digest algorithm, 1992.

[70] Chaitanya K. Ryali, John J. Hopfield, Leopold Grinberg, and Dmitry Krotov. Bio-inspired hashing for unsupervised similarity search. In *CVPR*, 2020.

[71] Jyotishman Saikia, Shihui Yin, Zhewei Jiang, Mingoo Seok, and Jae-sun Seo. K-nearest neighbor hardware accelerator using in-memory computing SRAM. In *ISLPED*, 2019.

[72] Ahamed Shafeeq and K. S. Hareesha. Dynamic clustering of data with modified k-means algorithm. In *ICICN*, 2012.

[73] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: a convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *ISCA*, 2016.

[74] Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 1948.

[75] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. WAN optimized replication of backup datasets using stream-informed delta compression. In *USENIX FAST*, 2012.

[76] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *USENIX FAST*, 2012.

[77] Nitish Srivastava, Elman Mansimov, and Ruslan Salakhutdinov. Unsupervised learning of video representations using LSTMs. In *ICML*, 2015.

[78] Secure Hash Standard. FIPS Pub 180-1. *National Institute of Standards and Technology*, 1995.

[79] Shupeng Su, Chao Zhang, Kai Han, and Yonghong Tian. Greedy hash: Towards fast optimization for accurate hash coding in cnn. In *NIPS*, pages 806–815, 2018.

[80] Jingdong Wang, Ting Zhang, Jingkuan Song, Nicu Sebe, and Heng Tao Shen. A survey on learning to hash. *IEEE TPAMI*, 2017.

[81] Guanying Wu and Xubin He. Delta-FTL: improving SSD lifetime via exploiting content locality. In *EuroSys*, 2012.

[82] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE TC*, 2015.

[83] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Michael L. Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Achieving human parity in conversational speech recognition. *arXiv*, 2016.

[84] Wayne Xiong, Jasha Droppo, Xuedong Huang, Frank Seide, Michael L. Seltzer, Andreas Stolcke, Dong Yu, and Geoffrey Zweig. Toward human parity in conversational speech recognition. *IEEE/ACM TASLP*, 2017.

[85] Qirui Yang, Runyu Jin, and Ming Zhao. SmartDedup: optimizing deduplication for resource-constrained devices. In *USENIX ATC*, 2019.

[86] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *USENIX FAST*, 2019.

[87] Xin Zheng, Qinyi Lei, Run Yao, Yifei Gong, and Qian Yin. Image segmentation based on adaptive k-means algorithm. *EURASIP JIVP*, 2018.

[88] Benjamin Zhu, Kai Li, and R. Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *USENIX FAST*, 2008.

[89] Han Zhu, Mingsheng Long, Jianmin Wang, and Yue Cao. Deep hashing network for efficient similarity retrieval. In *AAAI*, 2016.

[90] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE TIT*, 1978.