Synthetic Data Generation for Storage Trace Augmentation

LU PANG and KRISHNA KANT, Temple University, USA

Due to the increasingly data-intensive nature of the applications, the storage system performance continues to increase in importance and is often substantially responsible for the overall processing rate of the application. Fortunately, the storage technologies themselves are evolving rapidly in numerous ways from low-level read/write of bits from a device, all the way to the management of the entire storage hierarchy in large enterprise and cloud settings. Studying many of the important issues in this entire spectrum often requires storage access traces from the storage server side, but these are often hard to come by. To address this gap, we present a method to generate synthetic traces using a novel generative adversarial network (GAN) architecture that captures the realism and diversity of real storage traces. The generated traces can be used to augment the existing workload traces of interest for a variety of storage system studies. We demonstrate how the proposed method can generate storage traces that have the overall characteristics of the real traces and yet provide the behavioral diversity.

CCS Concepts: • **Information systems → Storage management**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Storage Traces, Generative Models, Synthetic Data, Data Augmentation, Time series

ACM Reference Format:
Lu Pang and Krishna Kant. 2024. Synthetic Data Generation for Storage Trace Augmentation. *ACM Trans. Storage* 1, 1 (April 2024), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

The increasingly data-intensive nature of data-centered applications coupled with the complexity of modern storage systems [3], requires that many storage system deployments, be designed and tuned to be as performant as possible. Storage systems not only have numerous knobs [28] to be tuned but also many dynamic data management policies that need to work consistently to deliver the lowest latency and highest data volume. In addition to being performant, the storage system also needs to avoid any data corruption or inconsistency. Such aspects are typically evaluated using *storage traces* obtained from the real storage systems. We note upfront that by storage trace we really mean the metadata trace (i.e., generating the block numbers that are read or written). We are not concerned with generating the block content per se. Some use cases such as deduplication do require block contents, but we do not consider those.

Collecting such traces usually has a substantial performance impact on the storage systems since the desired metadata must be extracted on each access, stored in a buffer, and occasionally flushed to the disk. Such overhead, coupled with privacy concerns regarding access patterns, has resulted in only a small number of traces being made available publicly, and most of those tend

This research was supported by NSF grant CNS-2011252 and an Intel grant.
Authors' address: Lu Pang, lpang@temple.edu; Krishna Kant, kkant@temple.edu, Temple University, 1925 N. 12th Street, Philadelphia, Pennsylvania, USA, 19122.

ACM 1553-3077/2024/4-ART
https://doi.org/XXXXXXX.XXXXXXX

to be of inadequate length. In contrast, many studies require quite long traces. Depending on the study objectives, the trace requirements may span from hours to weeks. For example, *tiering* is an essential feature for large scale storage systems, typically involving a slow but large hard disk drive (HDD) tier containing all the data, and a smaller/faster SSD tier for "hot" data. There may even be a third tier using emerging nonvolatile memory technologies for "very hot" data. Studying the performance of a tiering system adequately requires traces going over a week or more to properly represent daily workload patterns. Similarly, backup/restore performance requires long traces.

There are many tools that generate artificial storage traces [1, 2, 8]. Many of these are concerned with host systems' view of storage, typically in the form of access to files. Other trace generators focus on the storage system side and are concerned with accesses to blocks on a storage "Volume", which is the primary concern of this paper. As discussed in the related work section. Most tools generate traces geared toward specific applications. Our goal instead is to develop a general trace generator; one that doesn't need modeling of the specific application. It needs to be able to generate synthetic traces that can act as a drop-in replacement for the real trace, i.e., the generated trace should have similar overall properties as the real trace. *Yet, to be useful, it cannot be an exact or even approximate copy of the real trace. Instead, we should be able to generate traces on demand, where the specific characteristics of the real trace (e.g., regions of heavy or light traffic, read/write mixture patterns, activity in specific portions of the address space, etc.) appear based on the underlying patterns rather than simply duplicating what exists in the real trace. In short, we want the trace generation to be able to demonstrate diversity without deviating significantly from the overall characteristics of the real trace.*

To achieve these objectives, we explore artificial trace generation using Generative Adversarial Networks (GAN) [24]. GANs are well-explored in generating realistic images and short videos [6, 48, 54, 63] but their use in generating realistic storage traces poses several challenges that we discuss later in the paper. Thus the key contribution of this paper is the design of a specialized GAN that can accurately generate spatio-temporal features corresponding to the storage trace. Our model is able to deal with the large range and sparse nature of storage traces. Our design includes a mechanism to capture the distribution of the access frequencies in an attempt to seize the long-tail access frequencies found in real storage traces. A well-trained GAN model ideally will not memorize and duplicate the training data, and instead, it generates a varied set of episodes that is similar to the patterns found in the real traces.

The requirement of "diverse yet similar" trace generation makes the evaluation of the generated trace rather challenging. In particular, using a fine-grained evaluation based on matching between the real and generated traces over small time windows would not capture the diversity of the trace (e.g., moving averages, autocorrelation, etc.) and is thus not useful. At the same time, gross statistical measures on trace characteristics such as access frequencies and sizes, although important, are inadequate and thus we consider other measures as well. In this paper, we analyze the generated traces in comparison to real traces; including point statistics and the distribution of the access frequencies. Furthermore, we evaluated the generated traces using several measures to compare various aspects of the trace.

One such measure concerns the "heat prediction" developed in our earlier work [45]. The notion of "heat" is used in storage systems to determine what part of the data is actively read or modified by the applications. The synthetic traces will be used to train the model and will be evaluated against real traces.

We also use a model of storage trace similarity that we developed in [46]. It takes into account several aspects unique to storage traces including similarity across spatio-temporal decompositions. It compares crucial spatio-temporal features which are important for storage applications.

Additionally, we run the real and generated trace through a comprehensive SSD simulator with a write cache. We use this to compare the behavior of both traces. We examine the latency of the requests. We also compare the cache hit rate of both the real and generated traces. Most applications and hence the stored data do show a definitive locality of access in that much of the stored data is referenced only rarely while a small fraction of it is referenced heavily. This aspect has implications for many important operations such as tiering (i.e., movement of hotter data to faster layers of the storage hierarchy), caching of data (by the device, storage server, or the host), data prefetching, etc.

The rest of the paper is organized as follows. Section 2 introduces the related background work. Section 3 describes the challenges, data preparation, and essential details of our method. Section 4 analyses and evaluates the generated storage trace results. Finally, section 6 concludes the paper.

2 BACKGROUND

2.1 Emerging Storage Systems

Flexible and efficient storage systems form the backbone for working with big data and deriving intelligence from it. Storage systems continue to evolve rapidly along multiple aspects: underlying storage technologies, storage access protocols, storage management, and higher level abstractions such as file systems and how they map to the storage. In particular, while the traditional hard disk drive (HDD) technology continues to evolve rapidly to increase storage density [38, 40], it has also been largely augmented with the solid-state drives (SSDs) for primary storage (i.e., data that is actively used). The underlying NAND "flash" technology of SSD itself continues to evolve and provides a wide range of tradeoffs in terms of storage density, performance, cost, and endurance depending on the number of bits per basic "cell" of the technology. There are numerous other high performance emerging technologies that are beginning to fill the traditional large gap between storage and memory. Some examples of these technologies are Kioxia XL-flash, Intel Optane, Everspins's MRAM, etc. [17, 30, 31] These technologies naturally form a hierarchy, with HDD as the bottom layer (large capacity, slow, and cheap), SSD the middle layer (substantial capacity, fast, but more expensive), and emerging technology as the top layer (small, very fast and expensive). Often, the same basic technology may form multiple layers, such as slow vs. fast HDDs or SSDs. The entire hierarchy typically resides in a *storage server* which typically includes local DRAM cache and complex "volume management" to allocate requested space across one or more devices easily.

The basic unit of storage I/O is typically a "block" of size 4KB, addressed using a sequential number called LBA (Logical Block Address), and much of the I/O performance is usually quoted in terms of IOPS (Input/Output Operations Per Second), meaning block transfers/sec. A typical HDD can provide $\approx 25K$ IOPS for sequential transfers and as low as 1/100th as much for random transfers, while the latencies can be as high as $\approx$ 5ms (depending on the seek and rotational latency components). HDDs achieve higher bandwidth by striping data across many drives. As a result, HDD "farms" in data centers provide a large amount of storage capacity and overall throughput but still slow transfers. SSDs, in contrast, provide a latency in $\sim 100\mu s$ range and throughput of hundreds of K-IOPS to 1M or more IOPS). Higher speed technologies can support latencies under 10 μs and may support higher throughputs too. However, high speed technologies generally have a much higher cost, and thus much smaller installed amounts.

Modern enterprise storage systems can hold a large amount of data, ranging from tens of terabytes to hundreds of petabytes. Thus while the users may still access storage in 4KB blocks, using such a small size for storage management and transfer operations is extremely inefficient. Thus the storage operations often use a much bigger unit, usually known as a *chunk*. Chunk sizes may range from hundreds of KBs to multiple MB sizes. We will also use the notion of chunks in this paper, typically defined as 8MB.

Since the storage servers are removed from the hosts that use them, the view of the storage from the storage server side is radically different from the one of a host. A host only sees the storage allocated to it, usually in terms of the file system residing on it, and has no idea where the storage comes from or whether it is spread over multiple storage devices. In contrast, a storage server works with the *storage volumes*, each of which could be carved out of one or more physical devices. Although a storage volume could be used by multiple applications, generally, the major enterprise applications use separate volumes. In this paper, we are only concerned with the storage server side view of things, and storage traces obtained thus is a *block trace*, i.e., a list of accesses to the blocks, stamped by the time, starting LBA, operation (read/write), transfer size, etc. We also note that this paper does not specifically focus on object based storage systems, although we expect that the data access patterns should be similar to block accesses.

2.2 Data Augmentation

Data augmentation is a method to increase the amount and variety of data in a dataset, which has been a popular approach in machine learning and particularly in computer vision [9, 35, 56]. The new samples used to increase dataset size are generated from the existing data via a set of transformations, the appropriate transformations being application dependent. For example, in the case of images, it is easy to generate additional data by random cropping, translation, flipping, and adding some degree of noise. For time-series data, the suitable transformations are multiplicative scaling of signal amplitude (thereby making peaks more or less pronounced), stretching/compressing the time axis (to decrease/increase traffic density), scaling of mean and variance, and adding some random perturbation to it. This could be done in each dimension for a multi-dimensional time series.

Such operations create more diverse versions of the original data. If these variants are properly labeled, such data can be highly valuable in training machine learning models since it would force the model to learn more complex features and thereby avoid overfitting. However, creating additional data this way for driving a real system can be questionable; in particular, while highly diverse traffic episodes are good for stress testing, they are inappropriate for normal performance evaluation. At the same time, a highly constrained transformation may not introduce adequate diversity. In the GAN context, the two uses of transformation coincide – training the GAN on transformed data can improve the training but such training will also affect the generated traffic.

2.3 Generative Models

Generative models are statistical models that learn the data distribution and use it to generate samples of this distribution. This is in contrast with discriminative models, which are used to make a prediction based on the given samples.

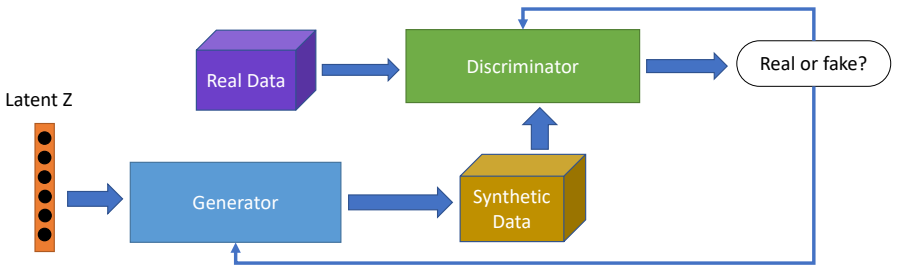


Fig. 1. Training Progresses of GAN

There are several methods that fall under the class of generative models. Variational Autoencoders (VAEs) are one such model. VAEs are autoencoders that consist of an encoder and a decoder. The encoder learns a distribution, commonly Gaussian, that can be used to generate a latent vector. The latent vector is a low-dimensional random vector that is used as a source of randomness for the generation process. The decoder learns a Gaussian distribution representing the distribution of data of the sample from the latent vector. During training, VAE learns the parameters for the probability distribution used. These probability distributions are used to generate a random sample that acts as a latent vector. The decoder is trained to take a latent vector, which is sampled from these distributions, and generates an output. One could interpolate the output of the encoder to generate data that contains features of the input. However, the requirement of Gaussian distribution assumes smoothness that is usually not present in storage traffic traces.

Diffusion models have emerged as a new class of generative models. Diffusion models start by gradually adding noise to the original data samples in a series of steps. They then learn to reverse the process, thus learning the model parameters needed to generate samples from noise. That is, given a sample x, at step $t-1$, the model will use a process to add noise to the sample at the next step, according to some noise distribution q. The task then becomes learning a distribution p that takes the sample with added noise x_t and can be sampled to generate x_{t-1}. To generate samples, the model will learn the joint distribution of the reverse process $p(x_{0:T})$. Thus, the diffusion models learn to generate samples from noise. However, these models tend to be rather slow and computationally expensive.

Generative adversarial networks (GANs) generate data via a transformation of a latent random vector into a data sample. The transformation is learned via the co-training of the generator neural network and a discriminator network. These two networks play a game where the discriminator optimizes for discrimination between the generated synthetic data (or "fake" data) and the real training data. Note that the generator and discriminator networks can be viewed as function approximators that are learned based on the given input-output value pairs [51]. Figure 1 shows the general architecture of GANs. Given a Generative network G, Discriminator network D, distribution of original dataset p_{data}, distribution of the random noise vector p_z, this can be written more formally as a min-max equation of value function $V(D,G)$ [23, 24].

$$\min_G \max_D V(D,G) = E_{x\sim p_{data}(x)}[\log D(x)] + E_{z\sim p_z(z)}[\log(1-D(G(z)))]$$

The first term encourages the discriminator to learn to maximize the value of real data x, where x is sampled from the original dataset. The second term encourages the discriminator to provide a low value for the data generated by the generator. Also, in the second term, the generator learns to produce data from a latent vector z that is sampled from the distribution of random noise. The generate learns by reducing the value that the discriminator outputs for generated data.

Ideally, the generator and discriminator come to an equilibrium where both the discriminator and the generator cannot do better. One danger of such a generative model is that the generator may get stuck into generating a random or specific pattern. This occurs when the generator starts producing one or a limited amount of samples that fools the discriminator, therefore, the generator has no force to improve the diversity of the samples. This is commonly known as *mode collapse*. There are many techniques [5, 55] to improve the stability of training GANs.

3 DATA GENERATION METHOD

3.1 Challenges in Storage Trace Generation Model

Using GANs to generate storage traffic is quite challenging as we need to capture complex correlations in time and space (block address). Other than avoiding mode collapse, we need to choose a

GAN architecture such that the model can be trained well without requiring an enormous amount of data or computing resources for training and execution. There is also the issue of diversity in the generated traffic as discussed earlier. GANs when trained allow us to draw samples from a distribution. The problem is that when the dimension size of the samples is large, GANs become difficult to train and require a substantial amount of compute/memory resources. In storage systems, the space dimension can easily range from millions to billions of chunks. Similarly, temporal variations in the traffic may cover many time scales, from milliseconds to weeks. It is not easy to generate realistic storage traffic while keeping both the data requirements for training the GAN and the training resources reasonable. This was essential not only for the feasibility of our current work but also to keep the mechanism practical in view of ever expanding scale and size of storage systems in the data centers. Towards this end, we have tried many different GAN architectures from the burgeoning GAN literature and their extensions. These include TGAN [53] and DCGAN [48], though the two that we explored in detail are BigGAN [12] and Progressive Growing of GANs (PGGAN) [33]. We dismissed the former as the basis of our model because it requires high resources (in terms of computational power and dataset size) to train. PGGAN progressively builds larger size traces till they reach the desired sample dimension size. Such an approach is more scalable, requires fewer resources to train, and can be more stable than attempting to generate traces directly at the original dimension. Thus, we develop our trace generation model based on PGGAN.

3.2 Data Preparation

We use several publicly available traces of enterprise storage servers. These generally provide 1-2 weeks' worth of block I/O traces, often taken from storage volumes dedicated to certain applications. (See section 4.1 for details). We preprocess the raw trace before we use it for our synthetic data generation model. The traces are data series of I/O accesses that contain the timestamp, offset, request size, operation (read/write), and often other information that we do not use. We then separate the trace into seven one-day long traces. We accumulate the data access frequencies in terms of fairly large size access units called *Chunks* over a time interval that we call timeslot t. In our experiments, each timeslot represents 150 seconds. This accumulation is done separately for read accesses and write accesses. We then represent the traces as data grids (successive time windows along the x-axis, chunk addresses along the y-axis, and the value being the number of accesses). Figure 2 shows an example of read and write accesses of the MSR dataset discussed later. The color represents the number of accesses, with a darker shade representing more accesses.

We then cut the read and write data grids into multiple data sample grids using the sliding window technique. Each data sample consists of m time slots. We concatenate the read and write data samples together on a new axis. The shape of each data sample is $2 \times n \times m$, where 2 is the axis that concatenates both read and write data samples and n is the total number of chunks on the server. The value of each element in the data samples varies a lot because of the large range of chunk addresses. Also, most of the values in the data samples are zero, since most chunks do not get accessed within m windows. This is typical of storage access; in fact, as the storage capacities and data set sizes increase, the actively used fraction is likely to go down. Therefore, we use a logarithmic scale to map the access frequencies to $[0, 255]$. We then normalize the data grids to $[0, 1]$. Mapping the initial frequencies through this process will reduce the training time of GANs and improve training stability. One reason for this is that the convergence rate will not be sensitive to the initial parameters of the model. Another reason is that large access values will not dominate the training and will allow the training process to update all the parameters on approximately the same scale.

We also generate downsampled versions of the data samples before starting the training so that the downsampling is done only once. The Pillow library [16] is used to perform the downsampling

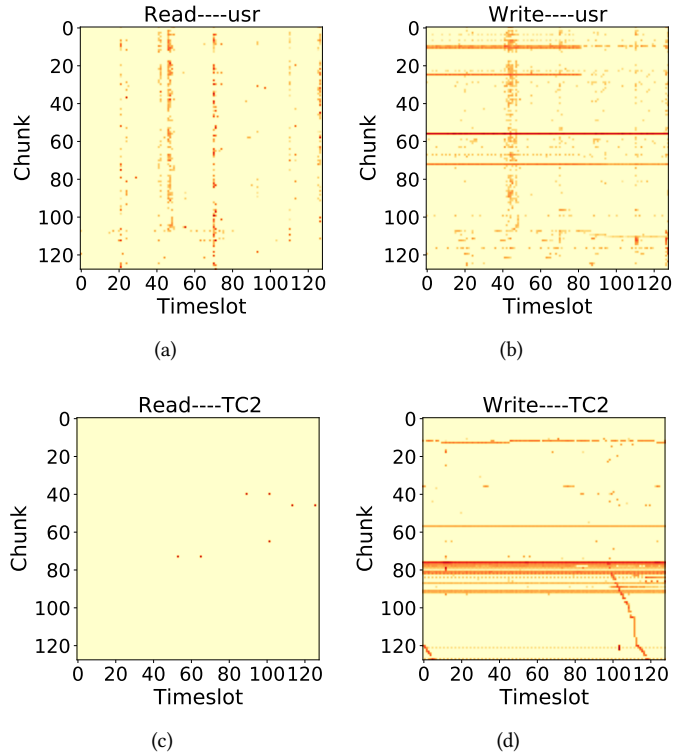


Fig. 2. Heatmap of read and write accesses of MSR (usr) workload and Tencent CBS (TC2) workload. For clarity, the first 128 chunks and timeslots are shown. The chunk size used is 8MB, and the timeslot size used is 150 seconds.

using bilinear interpolation. The downsampling results in fewer total number of chunks and timeslots, but the chunks and timeslots are larger. The usage of the downsampled data is described in the next section and its use in the generation process is shown in Figure 3. We denote the different sizes as $n_0 \times m_0, 2n_0 \times 2m_0, \ldots, 2^{k-1}n_0 \times 2^{k-1}m_0$, and $n \times m$, where $n = 2^k n_0$ and $m = 2^k m_0$. Here $n_0 \times m_0$ is the initial size of the data sample to start the training of our synthetic data generation model. Note that since n_0 is only 2^{-k} times n, the chunk size at the lowest level is 2^k times the chunk/timeslot size used in our original $n \times m$ representation.

3.3 Model and Training

Our model is a modification of PGGAN to scale the traces in progressive steps. That is, we start with a GAN that learns to generate a downsampled trace at the lowest level of $n_0 \times m_0$. Once that training phase is done, it adds an extra layer to the model which is trained to generate a larger trace. It repeats this process until it reaches the desired trace sample dimension size. The idea is to improve the stability of the training process and the quality of the generated data. The model builds on what it learned during the training of previous layers and learns to generate upscaled traces. After the model is trained, the model generates the synthetic samples using the final version of the networks which uses all the layers. The progressive training of PGGAN is shown in Figure 3.

Each new generator layer of PGGAN (shown as a green rectangle) connects to the previous layer and upscales the samples coming from that layer. Figure 4 illustrates the transition from data

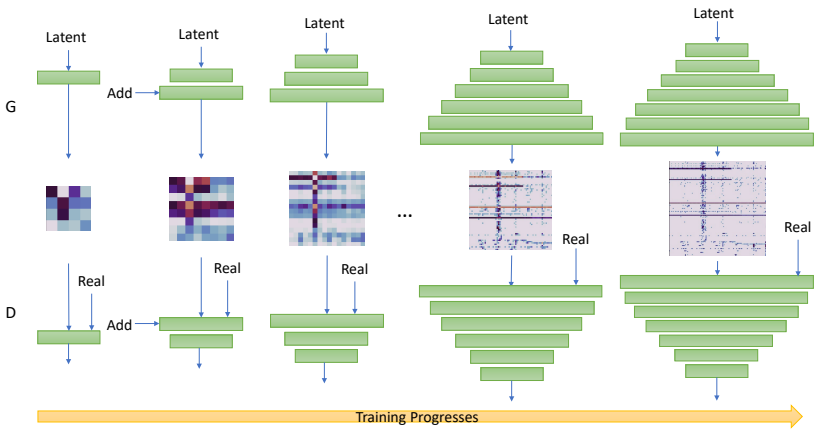


Fig. 3. As training progresses, new generator and discriminator layers are added. The new generator layer learns to generate upscaled data samples from the previous layer and the new discriminator layer is trained on the upscaled samples.

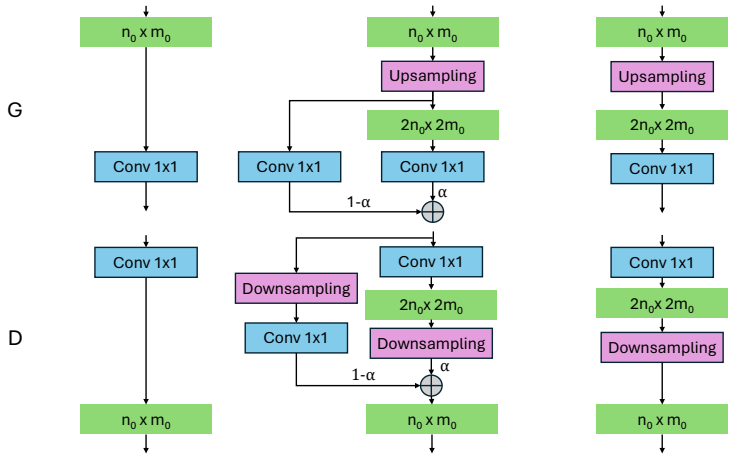


Fig. 4. Fading in the added new layers to G And D smoothly. When a new layer is added, the results of the previous layer are used ($\alpha = 0$), by upsampling its output, until the new layer starts getting trained. While the network is being trained, α is changed gradually. Once the new layer is trained, its output is used directly ($\alpha = 1$). "Conv" refers to Convolution layers.

samples with a dimension size of $n_0 \times m_0$ to data samples with a size of $2n_0 \times 2m_0$. It slowly adjusts a parameter α to linearly mix the upscaled output of the previous layer and the output of the new layer. That is, the output of the upscaled previous layer is weighted by $(1 - \alpha)$, whereas the output of the new layer is weighted by α. Initially, α is set to 0 so that the generated data in training comes from the upscaled sample and as training continues, it puts more emphasis on the new layer by increasing α linearly from 0 to 1. In this way, PGGAN makes use of the trained low layers to help optimize the parameters of the new layers.

Representation of a storage trace can be visualized as an image, as already shown in Figures 2(a) and 2(b). However, the considerations in generating images are very different than for generating realistic traces. In particular, with images, the key considerations are aesthetics and accurate

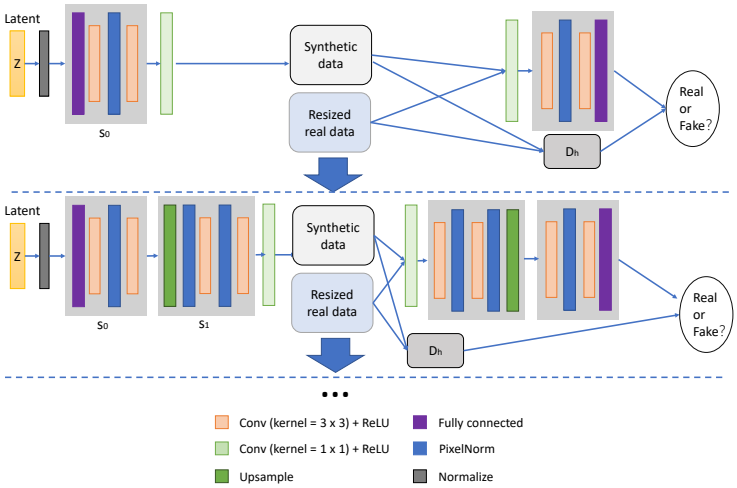


Fig. 5. Architecture of the synthetic storage trace generation model. Each progressive step adds common network blocks composed of the same set of layers as found in S_1. "Conv" refers to Convolution layers.

rendering of objects. With storage traces, we are interested in statistical characteristics such as achieving a similar distribution of accesses in addition to capturing prominent patterns such as frequent chunk accesses. What they both share is the ability to ensure diversity in the generated data. Since our model operates on storage traces, it works on sparse data. That is, at any point in time, most of the chunks do not receive any access. In order to better capture the features of the sparse data, we add another discriminator to guide the model to generate traces whose distribution is close to that of the real data.

Thus, our model consists of one generator network and two discriminator networks. One of the discriminators, denoted D, focuses on capturing the element-wise features. The other discriminator, D_{hist}, uses a histogram to capture the distribution features of the element values. A data sample x is input to D and D_{hist}, which output $D(x)$ and $D_{hist}(x)$ representing the probability of whether the input data is real or synthetic. The input of the generator network is a random vector z drawn from a uniform distribution. The initial output of G is the generated data of size $n_0 \times m_0$. We input both the down-sampled version of real data samples of size $n_0 \times m_0$ and generated synthetic data of size $n_0 \times m_0$ to train the discriminators to first differentiate between the real and generated data of dimension size $n_0 \times m_0$. Then we add layers progressively to G and D to generate data of dimension size $n \times m$. We do not add layers to D_{hist} since we build the histogram with a constant number of bins that cover the range of $[0, 1]$. Instead, we reinitialize the weights of D_{hist}. As training progresses, we keep on adding layers to G and D as well as reset the weights of D_{hist}. When we finish the training of the previous and move to the next dimension size. This process is repeated until we generate samples of size $n \times m$.

The architecture of our synthetic data generation model is shown in Figure 5. The architecture should be chosen carefully to take advantage of the structure found in the data for faster training and avoiding overfitting. We make use of Convolutional Neural Networks (CNN) [36]. CNNs use a sequence of convolutional layers, each of which convolves grid-like data with a *kernel* to learn its parameters. For example, each 3x3 kernel over the data would only require 9 parameters plus 1 parameter for a common bias. This is much more efficient than a fully connected (FC) layer for this type of data, where there will be a parameter between each element of the grid and each output of

the layer. Taking advantage of the structure in the data allows the model to learn more quickly, with less data, less number of parameters, and requires less memory to hold the parameters.

Our network grows as each layer is trained. The initial generator G consists of one FC layer, one two-dimensional (2-D) convolution layer with 3×3 kernel, and one 2-D convolution layer with 1×1 kernel. We first train the generator G to generate data of dimension size $n_1 \times m_1$ using initial input z. Then we add the replicated 3-layer blocks one by one to the G network, thus increasing the dimension size of the generated data incrementally. The 3-layer block is made of one upsample layer and two 2-D convolution layers with 3×3 kernel. The components of discriminator network D are mirrored layers of G. We use average pooling in D to downsample the data size. Average pooling does this by taking the average value of a window over the input, allowing us to reduce the size of dimensions of the of the input. The discriminator D_{hist} consists of one histogram construction unit, ten 1-D convolution layers, and one FC layer. The choice of convolution layers is based on their ability to find local correlation features in the data. By adding additional convolution layers, we increase the ability to capture longer range correlations. We use the FC layer to map the learned features.

In all three networks, G, D, and D_{hist}, every layer except the last uses the Leaky ReLU activation function [39]. These activation functions are needed in neural networks to introduce non-linearities into the network so it can learn non-linear mappings. ReLU outputs zero for inputs less or equal to zero and a linear function of the input for inputs bigger than zero. Leaky ReLU has a shallow slope for negative functions. Leaky ReLUs allow us to avoid dead neurons, where their output becomes zero and more training will not activate them.

The last layers of G and D use a FC layer. For G, it is to generate the data, and for D, the layer has one neuron which is used to label whether the data it received is generated or original. The last layer of D_{hist} uses a tanh activation function [32].

We initialize the bias parameters to zero and set all the weights to follow a normal distribution. We then scale the weights with the per-layer normalization constant to ensure the equalized learning rate [33]. We also perform element-wise normalization in G and D [33]. Each element represents the access frequency of a certain chunk during a certain time slot. We use Xavier uniform initialization [42] to initialize the weights in D_{hist} network in order to achieve convergence considerably faster. We also normalize the output of the histogram construction unit to stabilize the training.

The average pooling layer of size 2×2 is used in discriminator D to bring down the feature dimension size to half of the previous layer. We use the nearest neighbor upsample to double the feature dimension size in G. The convolution layers have 16 channels. As part of training, optimizers are used to adjust the parameters of the model, so as to reduce the loss. We optimize the synthetic data generation model using the Adam optimizer [34] which is often a first choice for training. We set the adaptive learning rate to 0.001. We use a batch size of 4 to train the data generation model. Our training starts with the sample of dimension size 4×32 and generates the final samples of size 256×2048, where 256 represents the number of timeslots in one trace sample and 2048 represents the number of chunks in one trace sample. We use a timeslot size of 150 seconds in our training. These parameters may need tuning for workloads that are significantly different from the workloads that we have used in our work. Our model has been trained with a dual-GPU machine with 11GB of GPU memory on the card and 64GB of main memory.

3.4 Loss Functions

The loss function provides the objective we wish to optimize. It is used in the training of the network by giving a value of how large our errors are. The larger the error the larger the parameter shift in the network during training. In the case of our model, the loss is composed of components. The

first part is the normal loss for GAN models which represents competition between the generator and the discriminator. The second part promotes the stability of training. The third part directs the network to train towards a certain behavior and induces capturing the features we deem appropriate for the storage trace generation task.

Binary cross-entropy loss: The goal of the discriminator is to distinguish between a real trace and a synthetic trace from the generator. The discriminator is trained to output a high probability for real data and a low probability for generated data. Then the binary cross-entropy is a measure of the discrepancy between the real and generated data. This value is used to train the discriminator so that it has a high probability for real traces and low probability values for synthetic data. The weights in the generator are updated so that it encourages the discriminator to give high values for synthetic data.

Wasserstein GAN with Gradient Penalty (WGAN-GP): WGAN [6] was developed to encourage stability of training a GAN and thus avoid mode collapse. It uses the Wasserstein distance, also known as the earth mover distance, as a measure between the generated probability distribution and the real probability distribution. WGAN-GP [25] was then developed to add a gradient penalty term to the loss function in order to enforce the Lipschitz constraint [10] on the discriminator. This ensures that the gradient is bounded by some constant. This prevents ever-increasing changes in the parameters and thus makes the training well behaved.

Differential histogram: In addition to the previous losses, we have added a differential histogram [62] that is fed into its own discriminator so as to capture the distribution of the accesses in the trace. The distribution in the traces is not Gaussian since the accesses are sparse and workloads in storage systems may have a multi-modal distribution.

4 EVALUATION AND DISCUSSION

In this section, we evaluated our synthetic storage trace generation method with various evaluation metrics. We used several publicly available block trace datasets in this evaluation and demonstrated that we can capture the essential characteristics of the trace in each case.

4.1 Publicly Available Traces and Characteristics

Our experiments were conducted with several block storage datasets coming from two sources. The first source is the Microsoft Research Cambridge (MSR) [43] dataset which contains one-week long block I/O trace of several different enterprise server workloads (using different storage volumes). We used the traces of the User home directories (*usr*), Hardware monitoring (*hm*), and Project directories (*proj*). Each trace also has the metadata of the associated disk from which the traces are collected from. We selected traces that are associated with one of the disks for our experiments.

The second data source is the Tencent production Cloud Block Storage (CBS) system [68]. The traces were collected from a production CBS system using a proxy server over 10 days. The proxy server is in charge of forwarding the I/O requests it receives from the client to the storage server. These traces were collected from a large number of cloud virtual volumes (virtual disks). The traces showed high variation in both the volume and access patterns over certain time periods. We evaluated several dozen of these traces and selected traces that represent the diversity of the dataset in the requested blocks and in the locality of requests. We randomly selected two of such traces and called these two workloads *TC1* and *TC2*. We used the weekday traces of the first week in our evaluation.

4.2 Evaluation

Figures 6-8 show the heatmap of synthetic traces and real traces for the MSR and Tencent CBS workloads. For clarity, only the first 128 chunks are shown. For the rest of the paper, the figures

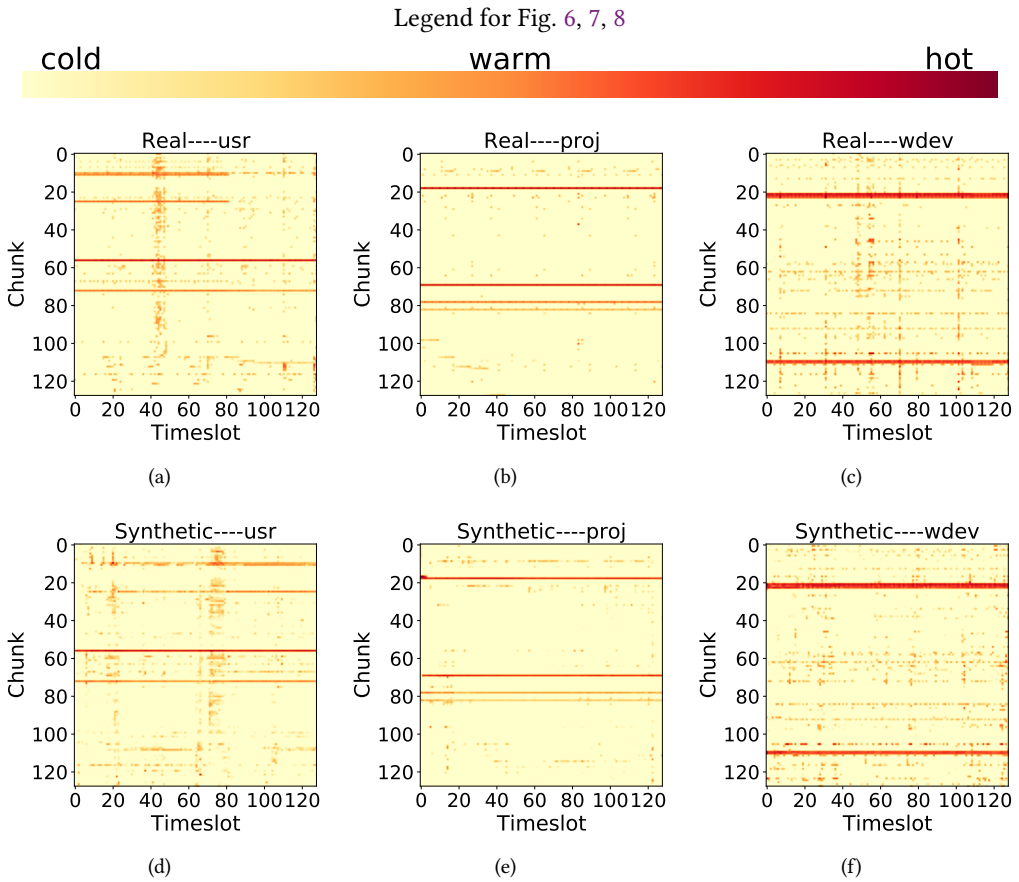


Fig. 6. Heatmap of real (left) and synthetic (right) write accesses of usr, proj, and wdev MSR workload. For clarity, the first 128 chunks and timeslots are shown. The chunk size used is 8MB, and the timeslot size used is 150 seconds.

will show the write accesses since most of the workloads are write-heavy. The generated traces show a similar behavior to the real traces but are not simply an approximate copy of them. To show this more clearly, the right panels in Figure 7 show the blown up view of the square regions marked up in the left panels. It is again seen that the right top and bottom panels are similar but not identical. It is also crucial to note that by design, the trace generator will pick random samples from the learned distribution, rather than trying to match the exact pattern to the real trace. Thus, *any given activity pattern (e.g., heavy accesses to some blocks) should not be expected to appear at a similar spot in the generated trace as the real trace.*

We started with a comparison of the statistics of the generated data compared to the real data. We used sixteen real trace samples and sixteen generated trace samples in our evaluation. The trace samples were randomly selected from the real and generated traces. The statistics were calculated for each of the sixteen trace samples. For each trace sample, we calculated the mean and standard deviation (S.D.) of the number of accesses over the chunks and the time range. These values differ across samples since different samples may include different patterns. Since there are many patterns and periods of high and low intensity in one trace sample, it is useful to ensure that these patterns are not skewing the overall number of accesses. Thus, we compared the maximum and minimum values of these samples. As we can see from Table 1 for both MSR trace and Tencent CBS trace, the

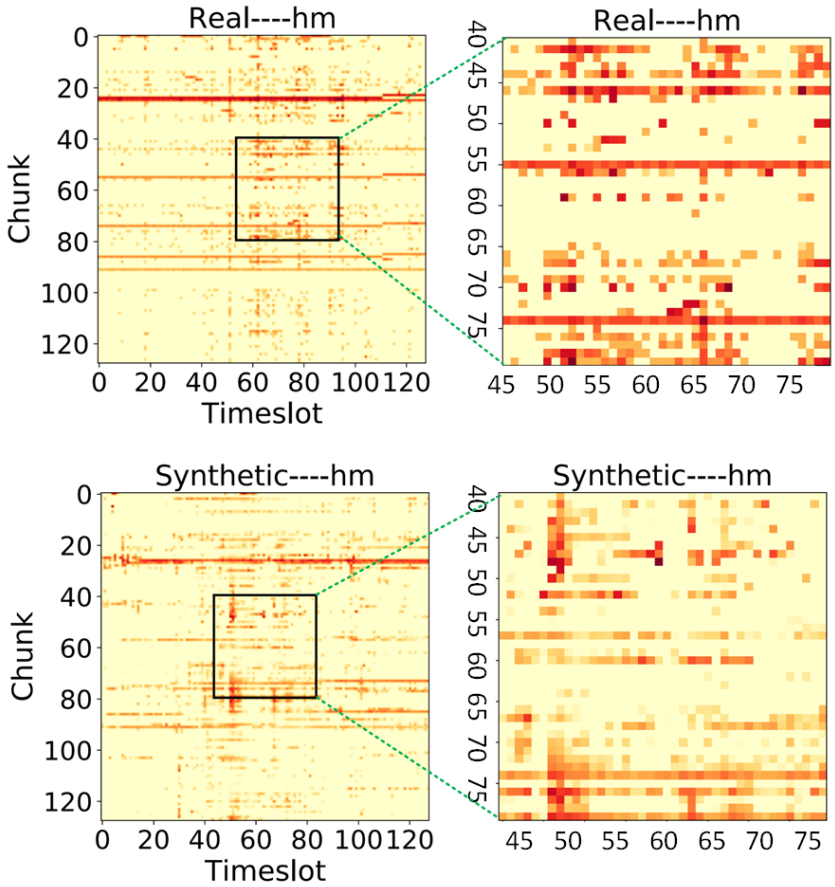


Fig. 7. Detailed heatmap of real (left) and synthetic (right) write accesses of hm MSR workload. For clarity, the first 128 chunks and timeslots are shown in the top two figures. The detailed heatmap of the rectangular section is shown in the bottom two figures. The chunk size used is 8MB, and the timeslot size used is 150 seconds.

Table 1. Mean and Standard Deviation (Real)

Statistics		**Workload**					
		usr	***proj***	***hm***	***wdev***	***TC1***	***TC2***
Mean	*Max (Real)*	1.44	2.06	3.48	1.21	7.13	3.36
	Max (Synthetic)	1.39	2.01	3.40	1.21	7.46	3.59
	Min (Real)	1.14	1.07	2.46	1.14	5.85	1.81
	Min (Synthetic)	1.07	1.05	2.58	1.09	6.30	1.82
S.D.	*Max (Real)*	12.21	16.25	17.59	10.27	26.94	17.12
	Max (Synthetic)	11.61	16.00	16.62	10.04	26.10	17.43
	Min (Real)	10.91	10.36	14.27	9.93	22.94	12.86
	Min (Synthetic)	10.38	10.11	14.32	9.53	23.44	12.75

max/min mean and the S.D. values of the generated trace samples are very close to the max/min mean and the S.D. values of the real traces. The reason that the mean is a low value is that the

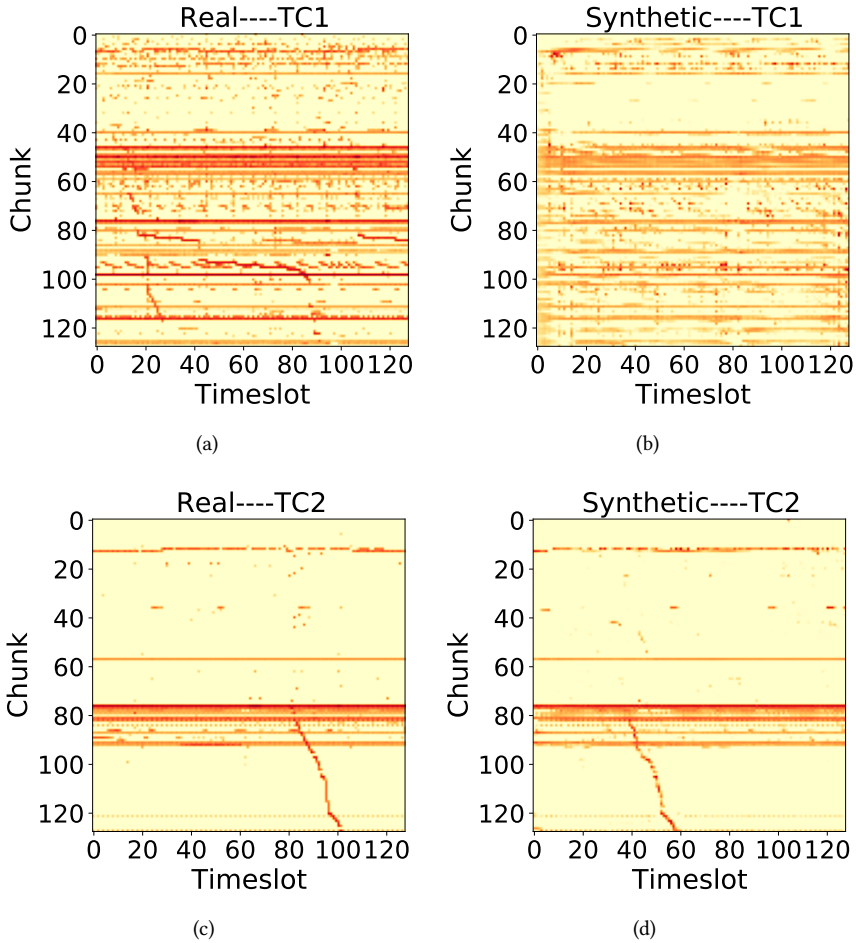


Fig. 8. Heatmap of real (left) and synthetic (right) write accesses of Tencent CBS workload. For clarity, the first 128 chunks and timeslots are shown. The chunk size used is 8MB, and the timeslot size used is 150 seconds.

accesses in the trace across time and chunks are very sparse. Generating overlapping means is in fact non-trivial because the tail distributions of the accesses are long.

For the reason stated previously, synthetic storage traces should also capture the distribution of the workload being targeted. That is, the patterns found in the synthetic trace should be similar to the patterns in the real trace. Figure 9 shows the histogram of the accesses for the MSR write workload and Figure 10 shows the histogram of the accesses for the Tencent CBS write workload. The closeness of the distribution shape between the real and synthetic traces validates the inclusion of the differential histogram loss as a way to capture the distribution of accesses.

Figure 11 shows the empirical cumulative distribution function for the access frequency of each chunk for all six workloads. The closeness of the real and synthetic empirical cumulative distribution function value also shows that our discriminator D_{hist} and the corresponding differential histogram loss is able to influence the model to generate samples that have similar distributions of access frequencies as the real samples. The smoothness of the synthetic cdf compared to the real cdf, is a result of the network attempting to learn the overall distribution via backpropagation. The

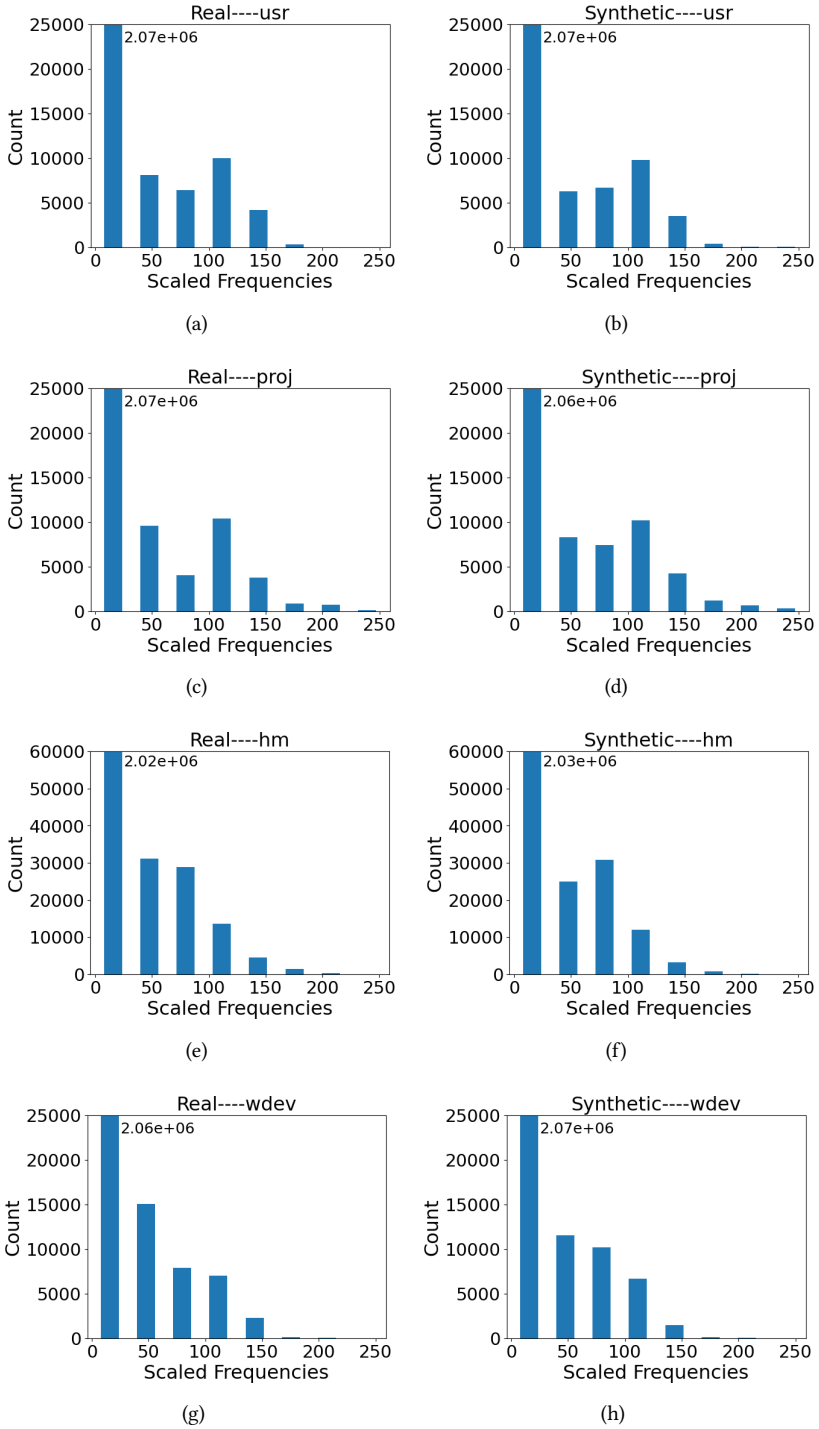


Fig. 9. Histogram of real (left) and synthetic (right) write accesses of Friday for MSR workload. The X axis represents the logarithmically scaled access frequency the chunks get. The Y axis represents the count of how many chunks get accessed at a certain frequency. For clarity, the Y axis is shown up to 25000 due to the long tail nature of storage requests. The range is divided into 8 equal buckets.

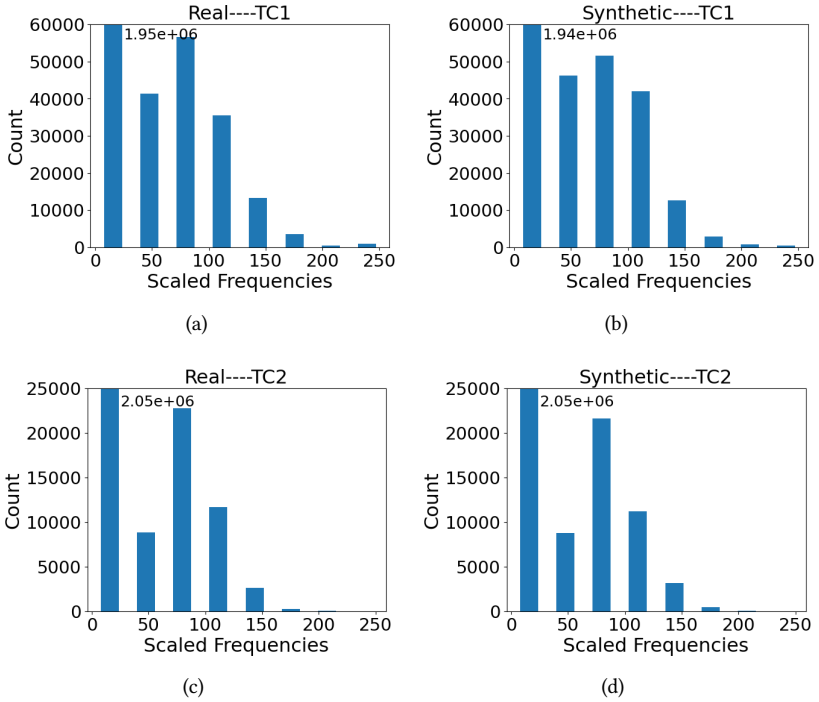


Fig. 10. Histogram of real (left) and synthetic (right) write accesses of Friday for Tencent CBS workload. The X axis represents the logarithmically scaled access frequency the chunks get. The Y axis represents the count of how many chunks get accessed at a certain frequency. For clarity, the Y axis is shown up to 60000 for TC1 and 25000 for TC2 due to the long tail nature of storage requests. The range is divided into 8 equal buckets.

generated access frequency can be a fractional number. In contrast, the real trace has a built-in discretization, possibly due to the OS accessing multiple blocks even when only one block is requested. For example, Linux often reads 4 or 8 blocks (16 or 32KB) when a single block is requested. It is certainly possible to alter the generated trace to conform to such discretization, which would make it look much closer to the real trace. However, we have not done so, since we wanted to focus upon the ability of the GAN rather than post-processing of the generated trace.

To evaluate the spatio-temporal behavior of the generated workload compared to the real workload, we used a storage traffic-specific similarity metric that we have introduced in our prior work [46]. We call this metric as Similarity Index for Storage Traffic (SIST). It generates three measures (main similarity S_M, activity difference S_A, and finer detail differences S_D). We used only the main similarity measure S_M, which factors in the spatio-temporal features of the two traces that we want to compare. This measure first decomposes the two traces using Discrete Wavelet Transform, and then computes the Dynamic Time Warping (DTW) distance for each spatial location in the two traces. Afterward, the overall Dynamic Time Warping distance across the spatial dimension is scaled with the total number of accesses in the trace.

In the following, we used the function $f_{SIST}(x, y)$ to represent the SIST measure between traces x and y. For example, the similarity between usr and hm workload samples will be indicated by $f_{SIST}(x_{usr}, y_{hm})$. The function $f_{SIST}(x, y)$ ranges from 0 to 1.0 with 1.0 meaning that the traces x and y have identical spatio-temporal properties. We expect samples from the same workload, for example, $f_{SIST}(x_{usr}, y_{usr})$, to have higher values than when compared to other workloads. We

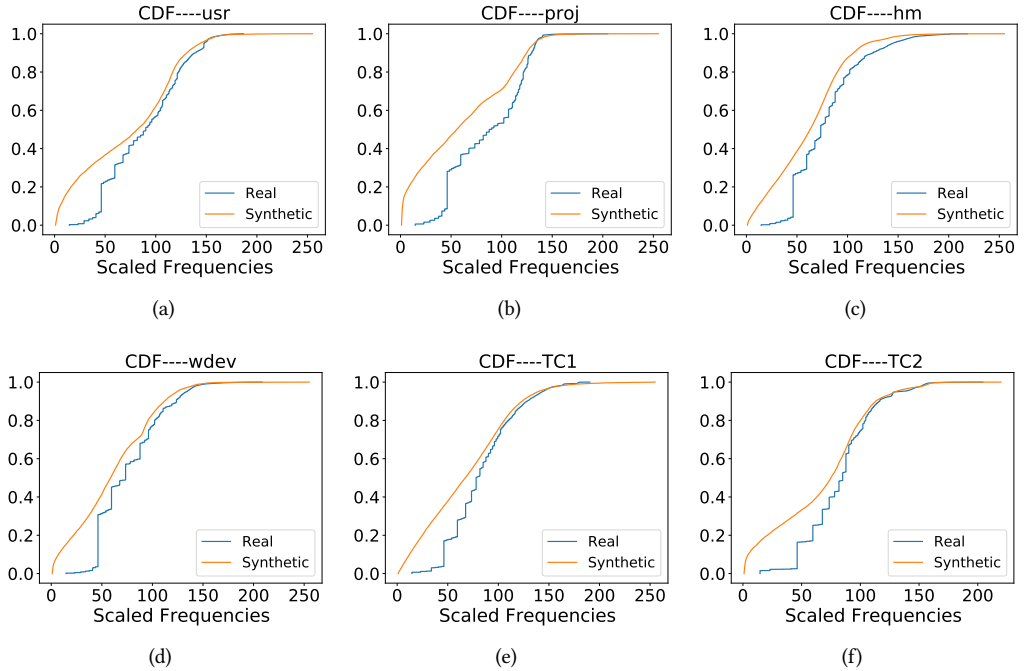


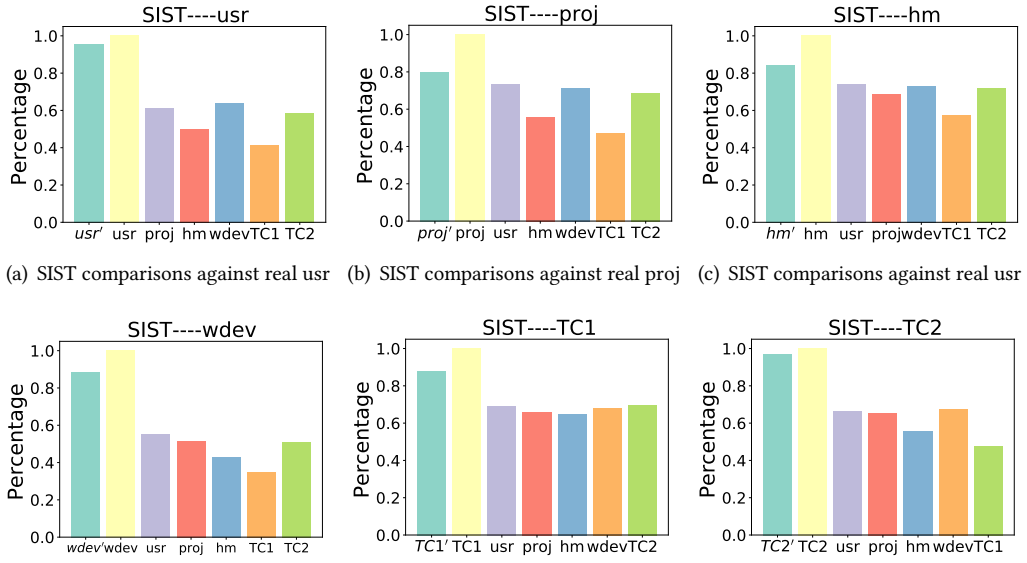
Fig. 11. Empirical cumulative distribution function for the access frequency of each chunk for all six workloads. The synthetic/real pairs are random samples from each workload. The x axis represents the logarithmically scaled access frequency the chunks get.

normalized the SIST values shown in Figure 12 using the SIST value of two randomly selected samples from the targeted workload which is noted in the title of each subfigure.

Figure 12 shows six bar charts, one for each workload as indicated. Each chart shows the SIST comparison results against the real traces for a target workload. For a specific target workload, the corresponding subfigure shows: the comparison results between the generated samples (as indicated in the chart label) and the original samples of the targeted workload, the original trace samples of the targeted workload against themselves, and the original traces of other workloads against the original traces of the target workload. For example, in Figure 12(a), we show the SIST measure between the generated and the real usr traces (marked as usr'). This is compared to the SIST results between each of the workloads and usr. It is seen that the SIST measure of usr' is closest to that of usr. The same behavior holds in Figures 12(b)-(f). For example, in Figure 12(f) TC2' is closest to TC2 as compared to others.

To further evaluate the behavior of the generated workload compared to the real data, we evaluated the synthetic trace on a prediction model. The model is a simplified version of a heat prediction model that we developed for storage traces [45]. The aspect of the model that we use in our evaluation is the heat prediction of the next time step based on the current activity. The heat prediction model predicts how many accesses each of the chunks gets for the next time step.

Fig. 13 shows the architecture of our heat prediction model. We used the Random Forest (RF) [11] composed of a collection of many Decision Tree (DT) [47] regression models. We used this collection of DT models to train over the input data and average the results of each model to produce the final heat prediction. We ran the prediction model with the mixed training dataset in that one out of five weekday data are randomly selected and replaced by the generated data. The heat prediction model used provides a prediction range of how many times a chunk will be



(a) SIST comparisons against real usr (b) SIST comparisons against real proj (c) SIST comparisons against real usr

(d) SIST comparisons against real wdev (e) SIST comparisons against real TC1 (f) SIST comparisons against real TC2

Fig. 12. Bar chart of normalized SIST for MSR and Tencent CBS workloads. Each graph considers one workload (e.g., "usr" in the first graph) as the baseline. The SIST values of all the shown workloads are relative to the baseline. The workload name without prime is the real traffic whereas the one with prime (e.g., usr') is synthetic.

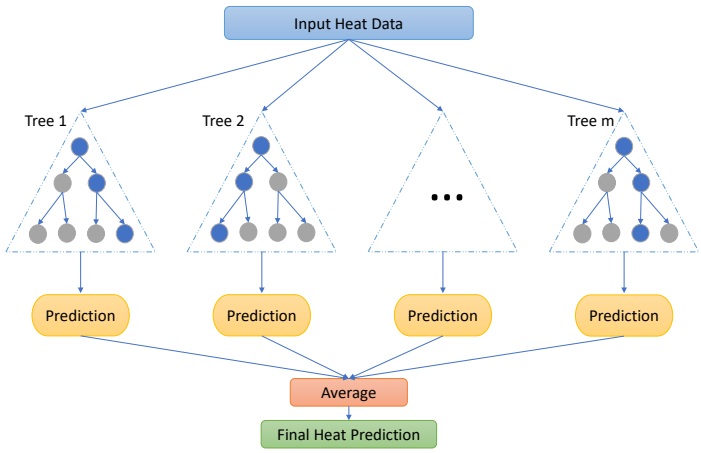


Fig. 13. Architecture of the heat prediction model.

accessed during the next time slot. If a chunk is accessed more than the inactive threshold, the chunks will be predicted as active. The active hit rate measures the ratio of the actual prediction results that fall within the prediction range. As we can see from Table 2, for most workloads, we achieved similar results for real and generated traces. As for the performance of "hm" workload,

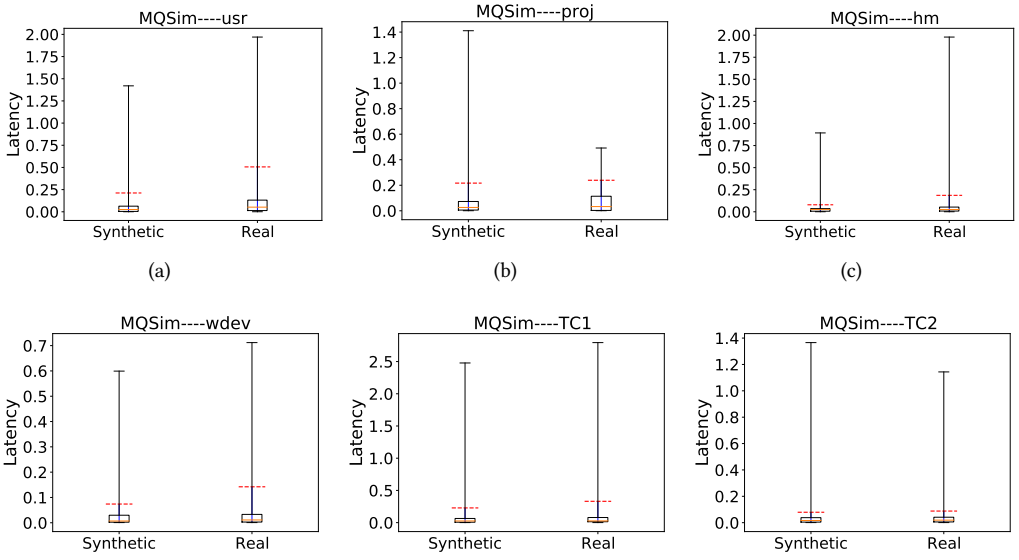


Fig. 14. Boxplot showing latency (ms) of a random synthetic and real trace run through MQSim. The whiskers cover the 99th percentile. The dashed line in red represents P90. The boxplot shows that the P90, and for most workloads, the P99 of the synthetic/real pairs within workloads are much closer than across workloads.

we can see that the synthetic and real traces of "hm" are closer to each other than the values of other MSR datasets (both synthetic and real).

Table 2. Active Hit Rate
Fraction of the prediction results that fall within the prediction range

	Workload					
	usr	***proj***	***hm***	***wdev***	***TC1***	***TC2***
Real	0.77	0.72	0.56	0.73	0.66	0.71
Synthetic	0.77	0.73	0.62	0.73	0.67	0.70

Table 3. MQSim Cache Hit Rate

	Workload					
	usr	***proj***	***hm***	***wdev***	***TC1***	***TC2***
Real	0.43	0.53	0.47	0.81	0.43	0.81
Synthetic	0.46	0.41	0.47	0.84	0.46	0.80

Another tool we used to show that the generated data from our method resembles real traces is an SSD simulator called MQSim [59]. We converted the synthetic output into a trace that can be used by MQSim. We randomized (uniform distribution) the specific LBA and time-of-request for each request in a chunk during a timeslot. MQSim was set up as an SSD of size 512GB with a write cache of 22GB and an overprovisioning ratio of 0.1. We ran the trace through MQSim and evaluated the trace using two measures. The first measure compares the distribution of the latency. In Figure 14, we present boxplots from random synthetic and real samples for each workload. We

can see that the median of synthetic/real latencies is low (due to caching) but the P90 for the same workload tends to have a similar spread and different spread between different workloads. For most of the workloads, P99 between the real and synthetic trace of the same workload also tends to be more similar. We also evaluated the write cache hit ratio shown in Table 3. The results show that for all six workloads, the cache hit ratio between synthetic and real is very close to each other.

5 RELATED WORK

5.1 Statistic property based generation

Many storage trace generators are currently in existence but can be classified into three broad classes. First, we have the simple ones which are usually driven by specified distributions for the size of stored files and access sizes, along with specifications of the extent to which the accesses are sequential, random, or mixed. FIO [8] and IOZONE [1] are two popular examples of such tools and they are largely used for benchmarking purposes [61]. Such generators are very limited in the traffic characteristics that they can emulate. Commonly applied statistical assumptions are sometimes not able to generate the trace accurately [20]. Atikoglu et. al. [7] use statistical modeling to model the generation of request properties, such as key size, value size, and inter-arrival rate for caching workloads. They showed their generated samples are similar to the original trace in terms of these statistical properties. In some works, it was shown that self-similarity [22] is a better model for synthetic trace generation. They used the performance measures that are provided by the disk simulator called DiskSim [21] to evaluate the distance between the synthetic workload and the real workload.

The second class includes the ability to generate ensembles of traces with different characteristics to emulate various workloads. They normally start with the real traces and then modify the real traces based on statistics. SPECstorage Solution 2020 [2] (and its earlier version called SPECSFS) fall into this category. Trace replay tools, such as hfplayer [26], are also included in this class. Hfplayer can take real traces, modify them in some way based on heuristics for inferring I/O dependencies or device characteristics, and replay them. They used several properties to evaluate whether the replay maintains the characteristics of the original trace workload such as the response-time and execution time. They also evaluate Type-P-Reordered metric which measures the number of requests that are reordered.

The third category attempts to generate realistic workloads using statistical models for specific applications or use cases. For example, Tarasov et. al. [58] proposed a method to generate realistic deduplication datasets by emulating duplication characteristics of the data that they analyzed. They used several statistical properties to measure their generated datasets, such as total files, total chunks, numbers of unique chunks and those that had one and two duplicates, directory depth, file size, and file type distributions. Other examples include generating specific benchmarks, such as TPC-H [67]. They used the response time provided by DiskSim to measure the accuracy of the synthetic workload. Ganapathi et. al. use statistical models to predict metrics for queries [19] and resource requirements for cloud applications [18]. The prediction model could then be used to generate workloads. To build this model, quality features need to be identified. They defined a predictive risk metric, which is similar to the R-squared metric, to compare the accuracy of their predictions. Wang et. al. [65] developed a simple model, called PQRS, to recursively decompose a trace into four regions and at each level capture the probability of access in each of those using an information theoretic approach. They evaluated their model through the comparison of the performance behavior (response time and queue length distributions) between the generated and the real trace. JEDI [52] is a trace generation tool for caching simulation that uses a custom traffic model called Popularity-Size Footprint Descriptor (pFD). The pFD captures properties that the

authors have deemed relevant for caching simulations. It is not clear how well it applies to other workloads. The quality of the generated trace is measured through object-level properties (i.e. object size, popularity, and request size) and cache-level properties (i.e. request and byte hit rates). For the object-level properties, they compared the distribution of these properties over the real and synthetic traces. For the cache-level properties, the hit rates are compared under various caching algorithms. Tarasov et. al. [57] also look at the problem of extracting essential characteristics of traces focusing, in this work, on block traces. They note, that in many cases, exact traces are not needed. Rather, it suffices to generate synthetic traces that capture the salient characteristics of the trace. The authors take a statistical approach, using multi-dimensional matrices to capture statistics; for example, a matrix can be designed to capture inter-arrival, read/write modes, and the I/O size. To deal with a shift in the statistical nature of the workload the authors developed a method called chunking; which captures the statistics over a shorter time period where they are stable. The method they developed can make a tradeoff between the size of the capture and the precision of the results. To characterize the accuracy, they measured the relative error between the synthetic and real traces on a set of trace statistics (e.g. read/sec, write/sec, request size, queue length, power consumption).

In this paper, we evaluate the quality of the generated trace through the two main approaches we find in the related works, namely statistical properties (e.g. access frequency) and the performance of applications (e.g. cache hit rate). In addition to these two approaches, we also evaluate the similarity of spatio-temporal patterns between the synthetic and real traces.

5.2 Data Driven based generation

When dealing with a large amount of data, capturing an accurate signature of a trace is useful to characterize a trace. Liu et.al. [37] developed IOSI, which is a three-phase approach to extract the I/O signature from noisy, zero-overhead server-side I/O throughput logs collected on supercomputers and job histories. They defined the I/O signature of an application as the I/O throughput generated by that application at the server-side storage of a given parallel platform during the application execution. The purpose of IOSI is to estimate the bandwidth needed for the user application. They first eliminated outliers during the Data Proprocessing Phase. Then they picked out the individual I/O bursts by wavelet decomposition during the Per-Sample Wavelet Transform. Finally, they identified common I/O bursts from multiple instances of an application's execution using CLIQUE which is a grid based clustering algorithm. They evaluated the quality of the I/O signature produced by IOSI and compared it to Dynamic Time Wrapping by cross-correlation and correlation coefficients.

Generative models have been used to generate network traffic [15, 50] but unlike storage trace, a network packet trace only has the temporal component (i.e., inter-packet times). They have also been used to generate continuous time-series data such as TimeGAN [66]. Medical signal generation such as EEG-GAN [27] for generating EEG signals has also been explored. However, medical signals have special features, such as periodicity with some inter-peak variations, and generally only small variations in the magnitude of peaks and valleys. Such characteristics should be represented accurately in the trace to make the signals realistic for a person. Storage traffic, in contrast, can have much more randomness and sudden shifts in the traffic.

Synthetic generation and augmentation using generative models have been explored extensively in the literature but mostly in the context of image generation. The works range from methods that allow the choice of classes for the outputs to augmentation by transforming the current dataset. However, the emphasis of the synthetic data generation and data augmentation works shown below differs from the goal that our work tries to accomplish.

Data Augmentation Generative Adversarial Networks (DAGAN) [4] is a model designed to generate class-specific data. It does this by learning the class parameters as part of its generator. This allows it to use few-shot learning and generate unique data from the distribution. DAGAN infers the class information from the example input and is able to generate output similar to the class.

Trabucco et. al. [60] studies diffusion models [29] to generate synthetic data to augment real data for downstream tasks. Their aim is to ensure the diversity of the generated data. For example, they note the issue of generating different species of animals. They propose an image-to-image transformation parameterized by pre-trained text-to-image diffusion models.

T-CGAN [49] uses the Conditional GAN (CGAN) [41] model to generate irregularly sampled time series data for data augmentation. CGAN introduces a way to condition the generator and discriminator by feeding the extra information, such as class labels or other domain knowledge, to both the discriminator and generator. CGAN is trained in this way so that data for specific classes can be generated on demand. T-CGAN generates time series, that are irregularly sampled, by conditioning the generator and discriminator with the timestamps of the time series.

Auxiliary Classifier GAN (ACGAN) [44] extends and improves on CGAN by introducing an auxiliary classifier to predict the class label of the generated data. Chen et. al. [13] adopt a data augmentation scheme based on ACGAN to directly generate different features of the desired acoustic scene with input scene conditions.

InfoGAN [14] learns interpretable representations to generate synthetic data with specific characteristics. InfoGAN introduces a classifier to maximize the mutual information between conditional variables and the generated data. This allows it to associate the conditional variable with the characteristics of the generated data. It does this in a completely unsupervised manner. This means that some features of the generated data can be controlled by changing a latent vector. But what features the individual element of the factor corresponds to are not known a priori. Wan et. al.[64] propose an InfoGAN based model to learn the coupling relations among bridge monitoring factors and then generate synthetic bridge monitoring data with various characteristics to augment the existing monitoring data.

6 CONCLUSIONS AND DISCUSSION

6.1 Conclusions and Extensions

In this paper, we presented a novel GAN based method for artificially generating block storage traces. The generation method is designed to capture the overall characteristics of the given real storage traces, and yet sports trace segments that show diverse characteristics. Such a "similarity with diversity" characteristic makes the mechanism useful for generating many instances of the trace. We also use several different methods to evaluate the quality of the generated trace including its statistical properties, SIST similarity, "heat" prediction performance, and predicted latency by SSD simulator.

One potentially useful aspect in generating storage traces is its specialization to certain categories such as heavy traffic, highly variable traffic, etc. We have not pursued this angle in this paper. Some simple post-processing techniques (e.g., those that scale the mean or variance or cause other systematic perturbations to the traffic) can accomplish such tasks without disrupting the correlation structure of the time series. Nevertheless, it is possible to train a generator that explicitly takes a class designator as input and generates traffic according to those characteristics. Many image GAN models have explored the generation of class-specific images such as the CGAN, ACGAN, and InfoGAN models discussed in the Related Work section. These methods could also be adapted to generate class-specific storage traces.

In the future, we plan to extend our technique to other spatio-temporal data such as data concerning vegetation, spread of tree and crop pathogens, or spatio-temporal variations in various socio-economic factors.

6.2 Limitations of the Generation Mechanism

Our method is concerned with storage-side (block) accesses, and does not deal with host-side (file) accesses. It may be possible to adapt the model for file-traces, but we have not explored that angle. Also, our focus is on access characteristics rather than the contents of what is accessed. Most available traces do not include block contents for obvious reasons, and most use cases only need the accesses. However, use cases like deduplication do need the contents.

GANs are generally difficult to train and require significant amount of resources to do so. Beyond the initial training, retraining may be necessary if the workloads change to the extent that many new sustained patterns emerge that the GAN did not come across during the training. It may also be necessary to tune the hyperparameters for environments that are very different from the ones considered here.

The GAN model only generates samples based on the prepared input data, i.e., the number of accesses to a chunk in a given timeslot. Within a timeslot, no further timing information is generated by GAN. It is possible to order and space out the accesses within a timeslot according to the temporal characteristics of the training data; however, that has not been a focus of this work.

It is important to note that the GAN traffic generation works by inputting a latent vector. After training, it is possible to input different latent vectors and generate multiple different instances of the traffic which sample from the learned underlying distribution of real traces. However, the relationship between the latent vector and the characteristics of the generated trace is extremely complex and not precisely controllable in many GAN based models. Nevertheless, one could exploit the latent vector dependence to generate many instances of traces with different characteristics. This is useful even though our architecture does not assign any sequential relationship between them.

As with any machine learning based method, the model can only learn from the patterns that it observes in the training data. It is expected that as the length of the training trace increases, it will contain more varied patterns that the GAN model can learn and thus reflect those in the generated traffic. However, exactly how long a trace is needed to capture all the patterns depends on the trace and is quite difficult to characterize. If the real traffic changes drastically following the training of the GAN model, the model would not be able to reflect that in the generated traffic. It is, however, possible to continue training the model in the background with recent traces, and thus be able to follow the traffic evolution.

ACKNOWLEDGMENTS

We are grateful to Dr. Jeremy Swift from Dell Corporation for a long-term collaboration on this work. His expertise and insights into the enterprise storage systems, gained from his 25+ years of work at Dell, were immensely valuable in ensuring that this work stayed in tune with the needs of designing and improving real enterprise storage systems. The initial part of this work was also funded by Dell, and we are grateful for the funding. The subsequent part of the work was funded by the NSF grant CNS-2011252.

REFERENCES

[1] 2016. Iozone Filesystem Benchmark — iozone.org. https://www.iozone.org/.
[2] 2020. SPECstorage Solution 2020 — spec.org. spec.org/storage2020/.

[3] Anis Alazzawe, Amitangshu Pal, and Krishna Kant. 2020. Efficient big-data access: Taxonomy and a comprehensive survey. *IEEE transactions on big data* 8, 2 (2020), 356–376.

[4] Anthreas Antoniou, Amos Storkey, and Harrison Edwards. 2018. Data Augmentation Generative Adversarial Networks. https://openreview.net/forum?id=S1Auv-WRZ

[5] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein GAN. arXiv:1701.07875 [stat.ML]

[6] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein Generative Adversarial Networks. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 214–223.

[7] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. 53–64.

[8] Jens Axboe. 2022. Flexible I/O tester. https://fio.readthedocs.io/en/latest/fio_doc.html.

[9] Philip Bachman, R Devon Hjelm, and William Buchwalter. 2019. Learning Representations by Maximizing Mutual Information Across Views. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/ddf354219aac374f1d40b7e760ee5bb7-Paper.pdf

[10] Yoav Benyamini and Joram Lindenstrauss. 1998. *Geometric nonlinear functional analysis*. Vol. 48. American Mathematical Soc.

[11] Leo Breiman. 2001. Random Forests. *Machine Learning* (2001). https://doi.org/10.1023/A:1010933404324

[12] Andrew Brock, Jeff Donahue, and Karen Simonyan. 2019. Large Scale GAN Training for High Fidelity Natural Image Synthesis. In *International Conference on Learning Representations*.

[13] Hangting Chen, Zuozhen Liu, Zongming Liu, and Pengyuan Zhang. 2020. ACGAN-based data augmentation integrated with long-term scalogram for acoustic scene classification. *arXiv preprint arXiv:2005.13146* (2020).

[14] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. 2016. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. *Advances in neural information processing systems* 29 (2016).

[15] Adriel Cheng. 2019. PAC-GAN: Packet generation of network traffic using generative adversarial networks. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 0728–0734.

[16] Jeffrey A. Clark. 2015. Pillow (PIL Fork) Documentation. https://buildmedia.readthedocs.org/media/pdf/pillow/latest/pillow.pdf

[17] Intel Corporation. 2023. Intel Optane Technology. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/optane-for-data-centers.html.

[18] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. 2010. Statistics-driven workload modeling for the cloud. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*. IEEE, 87–92.

[19] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 592–603.

[20] Gregory R Ganger. 1995. Generating representative synthetic workloads: An unsolved problem. In *Proc. Computer Measurement Group (CMG) Conference, Dec. 1995*.

[21] Gregory R Ganger, Bruce L Worthington, and Yale N Patt. 1999. The DiskSim simulation environment version 2.0 reference manual.

[22] María Engracia Gomez and Vicente Santonja. 2000. A new approach in the modeling and generation of synthetic disk workload. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No. PR00728)*. IEEE, 199–206.

[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems* 27 (2014).

[24] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[25] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. 2017. Improved Training of Wasserstein GANs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/892c3b1c6dccd52936e27cbd0ff683d6-Paper.pdf

[26] Alireza Haghdoost, Weiping He, Jerry Fredin, and David HC Du. 2017. On the Accuracy and Scalability of Intensive {I/O} Workload Replay. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 315–328.

[27] Kay Gregor Hartmann, Robin Tibor Schirrmeister, and Tonio Ball. 2018. EEG-GAN: Generative adversarial networks for electroencephalograhic (EEG) brain signals. *arXiv preprint arXiv:1806.01875* (2018).

[28] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. *Proceedings of the VLDB Endowment* 16, 7 (2023), 1712–1725.

[29] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising Diffusion Probabilistic Models. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 6840–6851. https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf

[30] Everspin Technologies Inc. 2023. Storage Solutions. https://www.everspin.com/storage-solutions.

[31] KIOXIA America Inc. 2023. XL-FLASH | Storage Class Memory (SCM). https://americas.kioxia.com/en-us/business/memory/xlflash.html.

[32] Barry L Kalman and Stan C Kwasny. 1992. Why tanh: choosing a sigmoidal function. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, Vol. 4. IEEE, 578–581.

[33] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. Progressive Growing of GANs for Improved Quality, Stability, and Variation. In *International Conference on Learning Representations*.

[34] Diederik P. Kingma et al. 2017. Adam: A Method for Stochastic Optimization. arXiv:1412.6980 [cs.LG]

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[36] Zewen Li, Fan Liu, Wenjie Yang, Shouheng Peng, and Jun Zhou. 2021. A survey of convolutional neural networks: analysis, applications, and prospects. *IEEE transactions on neural networks and learning systems* 33, 12 (2021), 6999–7019.

[37] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S. Vazhkudai. 2014. Automatic identification of application I/O signatures from noisy server-side traces. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (Santa Clara, CA) *(FAST'14)*. USENIX Association, USA, 213–228.

[38] Seagate Technology LLC. 2023. Everything You Want to Know About Hard Drives. https://www.seagate.com/blog/everything-you-wanted-to-know-about-hard-drives-master-dm/.

[39] Andrew L Maas, Awni Y Hannun, Andrew Y Ng, et al. 2013. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, Vol. 30. Atlanta, Georgia, USA, 3.

[40] Inc. Micron Technology. 2023. What is a hard disk drive (HDD)? https://www.crucial.com/articles/pc-builders/what-is-a-hard-drive.

[41] Mehdi Mirza and Simon Osindero. 2014. Conditional Generative Adversarial Nets. *CoRR* abs/1411.1784 (2014). arXiv:1411.1784 http://arxiv.org/abs/1411.1784

[42] Dmytro Mishkin and Jiri Matas. 2015. All you need is a good init. *CoRR* (2015).

[43] Dushyanth Narayanan et al. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*.

[44] Augustus Odena, Christopher Olah, and Jonathon Shlens. 2017. Conditional Image Synthesis with Auxiliary Classifier GANs. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 70)*, Doina Precup and Yee Whye Teh (Eds.). PMLR, 2642–2651. https://proceedings.mlr.press/v70/odena17a.html

[45] Lu Pang et al. 2019. Data Heat Prediction in Storage Systems Using Behavior Specific Prediction Models. In *38th IEEE International Performance Computing and Communications Conference (IPCCC)*. IEEE.

[46] Lu Pang and Krishna Kant. 2022. SIST: A Similarity Index for Storage Traffic. *Proc. of NAS confernce* (Oct 2022).

[47] J. R. Quinlan. 1986. Induction of decision trees. *Machine Learning* (1986). https://doi.org/10.1007/BF00116251

[48] Alec Radford, Luke Metz, and Soumith Chintala. 2015. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434* (2015).

[49] Giorgia Ramponi, Pavlos Protopapas, Marco Brambilla, and Ryan Janssen. 2018. T-cgan: Conditional generative adversarial network for data augmentation in noisy time series with irregular sampling. *arXiv preprint arXiv:1811.08295* (2018).

[50] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. 2019. Flow-based network traffic generation using generative adversarial networks. *Computers & Security* 82 (2019), 156–172.

[51] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533–536.

[52] Anirudh Sabnis and Ramesh K Sitaraman. 2022. JEDI: model-driven trace generation for cache simulations. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 679–693.

[53] Masaki Saito and Eiichi Matsumoto. 2016. Temporal Generative Adversarial Nets. *CoRR* abs/1611.06624 (2016). arXiv:1611.06624 http://arxiv.org/abs/1611.06624

[54] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, Xi Chen, and Xi Chen. 2016. Improved Techniques for Training GANs. In *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg,

I. Guyon, and R. Garnett (Eds.), Vol. 29. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2016/file/8a3363abe792db2d8761d6403605aeb7-Paper.pdf

[55] Tim Salimans, Ian J. Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. 2016. Improved Techniques for Training GANs. *CoRR* abs/1606.03498 (2016). arXiv:1606.03498 http://arxiv.org/abs/1606.03498

[56] Patrice Y Simard, David Steinkraus, John C Platt, et al. 2003. Best practices for convolutional neural networks applied to visual document analysis.. In *Icdar*, Vol. 3. Edinburgh.

[57] Vasily Tarasov, Santhosh Kumar, Jack Ma, Dean Hildebrand, Anna Povzner, Geoff Kuenning, and Erez Zadok. 2012. Extracting flexible, replayable models from large block traces.. In *FAST*, Vol. 12. 22.

[58] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 261–272.

[59] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. 2018. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *FAST*.

[60] Brandon Trabucco, Kyle Doherty, Max Gurinas, and Ruslan Salakhutdinov. [n. d.]. Effective Data Augmentation With Diffusion Models. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*.

[61] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P Wright. 2008. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)* 4, 2 (2008), 1–56.

[62] Evgeniya Ustinova and Victor Lempitsky. 2016. Learning Deep Embeddings with Histogram Loss. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 4177–4185.

[63] Carl Vondrick, Hamed Pirsiavash, and Antonio Torralba. 2016. Generating videos with scene dynamics. *Advances in neural information processing systems* 29 (2016).

[64] Ping Wan, Hongli He, Ling Guo, Jiancheng Yang, and Jie Li. 2021. InfoGAN-MSF: a data augmentation approach for correlative bridge monitoring factors. *Measurement Science and Technology* 32, 11 (2021), 114008.

[65] Mengzhi Wang, Anastassia Ailamaki, and Christos Faloutsos. 2002. Capturing the spatio-temporal behavior of real traffic data. *Performance Evaluation* 49, 1-4 (2002), 147–163.

[66] Jinsung Yoon, Daniel Jarrett, and Mihaela Van der Schaar. 2019. Time-series generative adversarial networks (TimeGAN). *Advances in neural information processing systems* 32 (2019).

[67] Jianyong Zhang, Anand Sivasubramaniam, Hubertus Franke, Natarajan Gautam, Yanyong Zhang, and Shailabh Nagar. 2004. Synthesizing representative i/o workloads for tpc-h. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 142–142.

[68] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 785–798. https://www.usenix.org/conference/atc20/presentation/zhang-yu