# Sift: Fast and Lightweight Docker Registries

Nannan Zhao[1], Hadeel Albahar[1], Subil Abraham[1], Ali Anwar[2], and Ali R. Butt[1]
[1]Virginia Tech, [2]IBM Research—Almaden

## Abstract

We did a deduplication analysis on around 50TB compressed image dataset and found that we can save about half of the storage space by doing file-level deduplication after decompressing all the images. However, deduplication affects Docker registry's performance especially for pulling performance because restoring layers require fetching files and compressing them into layers. To reduce deduplication restoring overhead(i.e., decompress/compression and additional networking and I/O), we propose a novel architecture that integrates compressed file-friendly deduplication and container user-friendly caching.

## 1 Introduction

Docker [3] containers have become a prominent solution for deploying modern applications because to their tight isolation, low overhead, and efficient packaging of the execution environment. Containers are running instances of Docker *images*. An image comprises a set of *layers* and is stored in a Docker *registry*. Each layer is a set of files that are compressed in a single archive. As layers are the building blocks of images, they can be shared among multiple images. A layer is uniquely identified by a collision-resistant hash of its content, called *digest*, and therefore no duplicate layers are stored in the registry.

Docker registries store a large amount of images and with the increasing popularity of Docker, registries continue to grow. For example, Docker Hub [4]—a popular public registry—stores more than 2 million public repositories that host one or more images. The number of public repositories grow by 1 million annually. Moreover, the cost of storing 60 TB worth of Docker images is around $7,000 dollars per month on Google cloud storage [9]. Then, the 1 million annual growth in the number of repositories amounts to about 130 TB, costing around $15,000 a month.

In our analysis of over 167 TB of uncompressed Docker images, we found that only 3% of the files are unique. Since many Docker images share many underlying layers, this considerable file-level redundancy across different images is not mitigated by Docker's existing layer sharing mechanism.

Deduplication is a fitting solution to eliminate redundancy in the storage system of container management platforms. However, existing deduplication techniques cannot be directly applied to the image storage system because of the unique characteristics of the Docker ecosystem. Images are stored as a collection of gzip compressed tar files, called layers. In contrast to uncompressed files, gzip compressed files have a lower deduplication ratio. A solution would be to decompress the layers before applying deduplication on Docker registries. However, restoring layers by assembling files incurs a considerable additional overhead on layer pull time and thus affecting the overall container startup time.

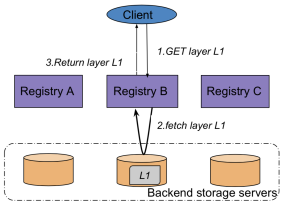


*Figure 1: PULL a layer from registry without dedup.*

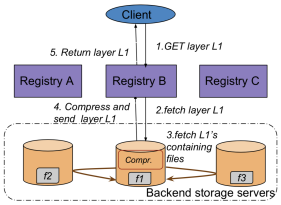


*Figure 2: PULL a layer from registry with dedup.*

Therefore, in this paper, we propose a low restore latency deduplication framework for Docker registries. Moreover, to improve layer pull performance, we design a user behavior based cache. The cache adaptively stores a certain amount of active users' layers, in a layer buffer, to speed up active users' pull time. The cache also selectively stores a few popular *shared* and *deduplicated* files, in a file cache, to accelerate the layer restoring process.

The rationale behind the user behavior based cache is that, instead of only focusing on layer-level access pattern, we consider the user behavior for cache replacement because users' behavior is more predictable than layer access pattern. Moreover, users are the ones who issue the layer/manifest pulls and pushes. For this reason, our cache replacement policy is an adaptive heuristic, top-down decision making process driven by users' behavior. During cache eviction, we evict the least recently accessed layer from a set of candidate layers exclusively referenced by the least recently active users to make room for new users' layers. After an inactive user's layer is evicted from the layer buffer, based on our algorithm, it can either be discarded or hosted in the file cache. For the latter, the layer will undergo offline decompression and file-level deduplication. For the incoming new users, we prefetch a number of their repositories along with their containing layers based on their access probabilities.

We evaluate our proposed design, Sift, by conducting a preliminary simulation-based study. Our study aims to explore the feasibility/benefits of deduplication and quantify the overhead it introduces. ■ **HA: add preliminary results**.

■ **HA: need to mention Figure 1 and Figure 2**

The organization of this paper as follows: We explain current deduplication practices in production registries and relevant Docker details in Section 2.2 and observations in Section **??**. We present deduplication analysis in Section **??**, and Siftdesign in Sectio **??**. lastly, we describe related work in Section 6, and conclude in Section 7.

## 2 Background, observations, and motivation

Docker [3] is a containerization platform to develop, deploy, and run applications inside *containers*. Docker container is a running instance of an *image* which wrap up an application and its runtime dependencies. Users interact with Docker using the Docker client which, in turn, sends commands to the Docker host. The Docker host runs a daemon process that implements the core logic of Docker and is responsible for *running* containers from locally available images. A Docker image consists of an ordered series of *layers*. Each Docker layer contains a subset of the files in the image and often represents a specific component/dependency of the image, e.g., a shared library. Layers can be shared between two or more images if the images depend on the same layer. Image layers are read-only. When users start a container, Docker creates a new *writable layer* on top of the underlying read-only layers.Any changes made to files in the image will be reflected inside the writable layer via a copy-on-write mechanism.

If a user tries to launch a container from an image that is not available locally, the daemon `pulls` the required image from the Docker registry. Additionally, the daemon supports `building` new images and `pushing` them to the registry.

### 2.1 Docker registry and backend cloud storage systems

The Docker registry [4] is a platform for storing and distributing container images. It stores images in *repositories*, each containing different versions (*tags*) of the same image, identified as `<repo-name:tag>`. For each image, the Docker registry stores a *manifest* that describes, among other things, which layers constitute the image. The manifest is a JSON file, which contains the runtime configuration for a container image (e.g., target platform and environment variables) and the list of layers which make up the image. Layers are identified via a digest that is computed as a hash (SHA-256) over the uncompressed content of the layer and stored as compressed archival files.

Although Docker registry is a layer-level content addressable storage system holding all the images, it delegates storage to drivers which interact with either a local file system or a remote cloud storage like S3 [**?**], Microsoft Azure [**?**], OpenStack Swift [**?**], and Aliyun OSS [**?**]. For example, Google Container Registry [11] use cloud storage as their backend Docker image storage systems. Users `push` and `pull` Docker images to and from their repositories stored on cloud storage. To facilitate a fast and high-availability service, container registries use regional private repositories across the world. This geographical distribution allows users to store images close to their compute instances and experience a fast response time. For example, IBM's Container Registry setup spans five regions [20].

### 2.2 Observations and Motivation

**Deduplication for Docker registries?.** On-cloud global deduplication software is widely adopted by cloud enterprises for reducing cloud storage consumption and overall storage cost. For example, StorReduce [**?**], the deduplication software choice of Google cloud and AWS, performs in-line data deduplication transparently and resides between the client's application and the hosting cloud storage. A number of deduplication methods focus on client-side data deduplication to ensure that only unique files are uploaded, to save network bandwidth, by having the client send a duplicate check request [**?**] [**?**]. For example, xxxx■ **Nannan: Hadeel, can you add one example and few related-work citation?**.

We observed that in Docker Hub, the number of public repositories is constantly increasing with a growth that amounts to around 1 million repositories annually. Based on our estimation this will cause at least 130 TB of annual growth in storage needs. Intuitively, the above deduplication techniques can be applied to eliminate redundant data from the Docker image storage system. Except, the Docker image dataset is different from the common data stream. They are *compressed archival files*.

We performed an in-depth and empirical analysis of file-level deduplication on container images. We analyzed around 50 TB of compressed Docker images (totally 170TB after decompression) collected from Docker Hub. Surprisingly, only around 3% of the those files are unique while the rest are redundant copies. After removing the duplicate files, we can save around half of the storage space.

To eliminate these redundant files from the compressed layer files, changes must be made to these deduplication methods. Such changes should recognize the compression formats, perform decompression before feeding the data to a block-level or file-level deduplication process. Otherwise, the deduplication ratio would be very low since compressed files have a very low deduplication ratio [**?**]. Moreover for restoring a layer, the

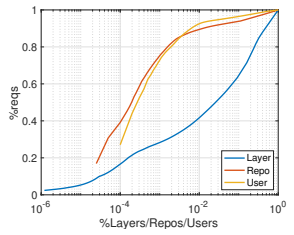


*Figure 3: Popularity of layers, repos, and users.*

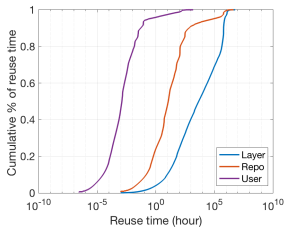


*Figure 4: CDF of reuse time for layers, repos, and users.*

layer's containing files should all be fetched from wherever they reside, as they are likely to be scattered across multiple servers, and compressed together, which causes a considerable overhead on pull layer requests performance.

**Use registry as a web/proxy cache?.** Traditionally, caches are placed as close to the requesting client as possible, such caches are known as proxy caches, or web/HTTP caches for the short-lived storage of frequently requested/accessed data to reduce the server's lag. For example, ■ **Nannan: Hadeel, just like storage and dedup parts, this cache part we just needs two examples. with name. others we can put in citation** Docker registry is a web server that serves docker pull and docker push requests. Intuitively, registries can be deployed as a proxy cache to host frequently requested layers to speedup image pulls and improve performance.

We analyzed IBM container registry workload in Dallas(`dal`) [20] and made the following observations.

## 2.3 Need for User Behavior based Cache Management

The following observations are obtained by analyzing the Dallas(`dal`) registry workload collected from IBM Container Registry over the course of 75 days [20].

**Requests to layers are heavily skewed but layer reuse time is very long.** Figure 3 shows the registry accesses to layers and repositories, and accesses by users. Layer accesses are heavily skewed. For example, 25% of popular layers account for 80% of all requests. For repository accesses and accesses by users, the skew is more significant than it is for layers. 94% of all requests are accessing only 10% of the most frequently accessed repositories and only 9% of users, most active ones, issued 97% of all requests. This means that only a few extremely active users create their repositories in the `dal` registry and issue the majority of requests to the registry.

Figure 4 shows the reuse time of layers and repositories, and reuse time by users. Reuse time is the duration between two subsequent requests to access the same layer or repository while the reuse time by user is defined as duration between two subsequent requests issued by the same user. The layer reuse time is long. The median

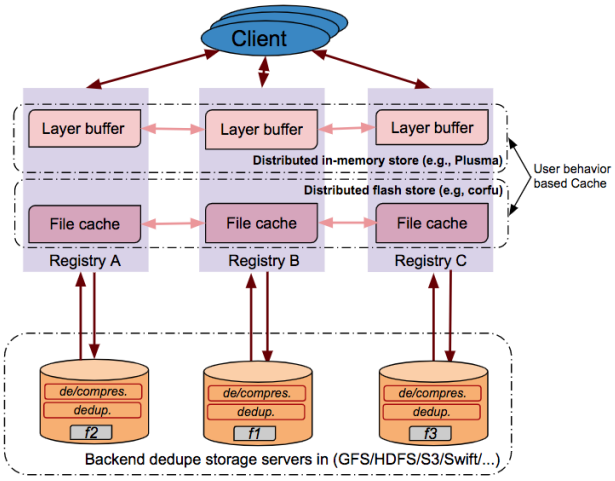


*Figure 5: Architecture of Sift.*

reuse time of a layer is 1.3 hours. 80% of repositories experience the highest request frequency, with a reuse time of around 2 minutes. 90% of users remain active for at least 0.06 seconds. In other words, most of the layers stored in the registry are not frequently requested in a very short time period while users remain active for a longer time. This is because users request different layers or manifests.

**User active time is predictable.** Based on the above observations, we believe that the user's active time is predictable. By just maintaining a LRU list of users, we achieved 99% accuracy for predicting user active time. For this reason, we utilize this predictability in our cache algorithm.

# 3 Sift Design

## 3.1 Integrating deduplication, caching, and Docker registries

■ **Ali: Following text makes no sense.. what are you trying to explain??** ■ **Nannan: This part includes our architecture. Our architecture doesn't required many words to explain**

Sift seamlessly integrates the management of cache, deduplication on backend storage system (called backend dedup storage), and Docker registries ■ **Ali: Following information should be part of Background section.. Intuitively, registries can be deployed as a proxy cache to host frequently requested layers. This approach, a registry as a pull-through cache, speeds up image pulls and improves overall performance. At the same time, the backend cloud storage can leverage deduplication to save storage space.** ■ **Nannan: moved** and solves the unique problems concerning the integration of caching and deduplication to the Docker registry. First, for caching layers, `pull` layer requests are difficult to predict because same layers are not accessed frequently. We observed that, around half of the

cached layers' are not accessed again for at least 1.3 hours which means that if we cache a layer, we might need to wait hours to get a hit on that layer as mentioned in 2.2. This is because when a user pulls an image from the registry, the Docker daemon on the requesting host will only pull the layers that are not locally stored. ■ **Ali: I do not understand the following sentence.** Ali Anwar et al., proposed a prefetching method [20] based on the `push-pull` relationship: when there is a `PUSH` layer request directly followed by a `GET` manifest request, a `GET` layer request will most probably follow. However, based on our trace■ **HA: we have to mention which trace???** analysis in 2.2, only half of the `GET` layer requests have a precedent `PUSH` layer request within the trace collecting duration of 75 days. This means that, after a user pushes a layer to the registry, it takes a few days, weeks, or even months for a user to make a `pull` request. ■ **Ali: The above statement is incorrect. You have to distinguish between GET layer requests that are issued after a (PUSH layer + GET manifest) request and a normal GET layer request. FAST paper only talk about case 1. Whereas you are generalizing that any GET layer request should have a precedent GET layer request which is wrong. We can make a case that not all GET layers requests have a precedent PUSH layer request but we can not say that it takes a few days, weeks, or even months for a user to make a pull layer request after a push layer request.** ■ **Nannan: I mean the first case, push beyond your trace collection time.** Second, we can not deduplicate compressed layers. Hence, for deduplication each layer needs to be uncompressed and then undergo a file-level deduplication. Similarly, to restore a layer, we need to fetch files from multiple servers and then compress them in to a tar file. This whole process can cause a considerable overhead on `pull` layer requests performance. Deduplication also slows down `push` layer requests because its highly demand for CPU, memory, I/O, and network resources. ■ **Ali: Explain how push layer requests are not effected?**■ **Nannan: fixed**

## 3.2 Design

To meet these challenges, we propose a new registry design featuring a user behavior based cache to reduce the performance degradation caused by deduplication on backend storage system (Figure 5). Based on our observation, user's active time is easier to predict as shown in Figure 4. Our cache design considers user behavior such as when a user is most likely to be active for layer evictions from the cache.

Considering that layer size is around several megabyte on average [20], a small main memory cache cannot accommodate many active users' layers. To address this issue, we couple main memory and flash memory to provide separate caching for layers and *deduped* files. We call compressed layer cache cache and *deduped* files cache, *layer buffer* and *file cache*, respectively. For handling cache evictions, we first evict inactive users' layers from the layer buffer. Next, we *dedup* the evicted layers, then store the *deduped* files into the file cache (detailed in 3.3). When a user requests a layer not present in the layer buffer, the request is forwarded to the file cache (detailed in 3.3). If a layer is not found in both the layer buffer and the file cache, the request is forwarded to the backend dedup storage system. Note that during layer deduplication, uncompressed layer files are scattered across multiple servers. We call all the files belonging to a layer on a specific server a *slice* of the layer. All the slices for a layer are fetched in parallel for performance improvement. Each backend server compresses the slice and directly send the compressed slices back to the user. We modify the Docker client interface such that when it receives all the compressed slices, it can decompress them into a single layer. Furthermore, compressing slices of layers in parallel considerably mitigates the compression latency caused by compressing a whole layer since compression time depends on the size of the uncompressed data.

## 3.3 Operations

The Docker registry API is almost the same as the original registry. The user's interaction with the Docker client is unchanged. The user simply pushes and pulls images to and from the registry. In this subsection we explain Docker operations integration with Sift.

**Push.** After receiving a `push` layer request from the client, Sift first buffers the layer in the layer buffer for later accesses. The layer buffer can be implemented on a distributed in-memory store, e.g., Plusma. Meantime, Sift will also submit this layer to the backend dedup storage system.■ **Nannan: dedup** Our layer buffer and file cache use write through policies. Since there is no modification to the layer or files, there is no data consistency issue between the cache and the backend dedup storage system. A cold layer eviction might be triggered on layers stored in the layer buffer. The cold layer will be evicted to the file cache, but first it will be deduped and then stored in file cache. The deduplication process includes the following steps that are applied on every victim layer evicted from the layer buffer to the file cache:

1. decompress and unpack the layer's tarball into individual files;
2. compute a *fingerprint* for every file in the layer;
3. check every file's fingerprint against the *fingerprint index* to identify identical files already present in the file cache;
4. only store the unique files in the file cache and up-

date the ■ **Nannan: unique file** *file index* with unique files' fingerprints and file location along with its host address;
5. create and store a *layer recipe* that includes the file path, metadata, and fingerprint of every file in the layer;
6. remove the layer's tarball from the layer buffer.

Layer recipes are identified by layer digests and files are identified by their fingerprints. These identifiers are used to address corresponding objects in the underlying flash storage. *fingerprint index* and *layer recipes* ■ **HA: why are they in italic again?** are stored on Redis [15].

File cache is a flash-based distributed cache that can be implemented on distributed log structured store, e.g., CORFU. The unique files are flash-friendly because there is no modification to these files. The eviction of a cold layer from the layer buffer might also trigger the eviction of its files. And since the backend dedup storage system already stores a backup of the layers, we can simply discard the victim files from the file cache. The cache replacement algorithm is presented in section 3.4.

**Pull.** A `pull` layer request that finds its desired layer in the layer buffer is a layer buffer hit. Otherwise, if it finds its containing files in file cache, that is a file cache hit and Sift has to *reconstruct* the layer from the file cache based on the layer recipe. Sift performs the following steps *inline* for restoring a layer from the file cache:

1. find the layer recipe by the layer digest and get the layers' containing files' *fingerprints*;
2. lookup the *fingerprints* in *fingerprint index* to get a destination server list.
3. forward the `pull` layer slice request and layer recipe to each server in the server list.

Once the `pull` layer slice request is received, each destination server will initiate a layer slice restoring process which performs the following steps:

1. Prepares a directory structure for the layer based on the layer recipe;
2. Copy the locally available files into the directory tree,
3. Compresses the layer's directory tree into a temporary tarball;
4. Send the layer tarball back to the client and then discard the layer tarball.

If A `pull` layer request is miss on both the layer buffer and the file cache, the request will be forwarded to the backend dedup storage system. The layer restoring process on the backend storage system is similar to restoring from the file cache. Many modern storage systems with the deduplication feature can be used as our backend storage system, including GFS [**?**], HDFS [12], S3 [**?**], and Swift [**?**]. We can modify the above systems so that they can recognize the compressed layer file type and decompress them before performing deduplication. Moreover, the systems can restore layer slices in parallel.

■ **Nannan: put in background or intro, conclusion, discussion?** At a high-level, registries act like a distributed cache siting closer to clients to provide better performance in terms of response time. The layer buffer holds hot layers that belong to active users. The utilization of flash memory to store unique files not only mitigates the capacity limitation of the main memory cache, it also offers fast random read accesses.

## 3.4 User-based Cache Algorithm

**Algorithm 1:** User-based cache replacement algorithm.

**Input:** $S_{thresh}$: Capacity threshold for layer buffer to trigger eviction.

**while** *free_buffer* $< S_{thresh}$ **do**
  $last\_usr \leftarrow UsrLRU.last\_item()$
  **for** *layer in reversed(LayerLRU.items())* **do**
    **if** *layer exclusively belongs to last_usr* **then**
      **Evict** $LayerLRU[layer]$
      $free\_buffer += sizeof(layer)$
    **end**
  **end**
  **Evict** $UsrLRU[last\_usr]$
**end**

We use our observations of the user access pattern to guide our cache replacement since the user's active time is more predictable as discussed in Section **??**. In 3.1, the incoming `push` layer requests are first buffered in the layer buffer and later evicted to the file cache. If there is a `pull` layer request miss in both the layer buffer and the file cache, Sift will fetch the layer from backend storage system and store it in the layer buffer. When a cache pressure happens in both the file cache and the layer buffer caused by shortage of free space. The layer buffer or the file cache will simply evict or delete some layers or files and reclaim space. Since our layer buffer and file cache both share the same cache replacement algorithm, we only present our user-based cache replacement algorithm for the layer buffer as shown in Algorithm 1.

Free space in the layer buffer is too low if ($free\_buffer < S_{thresh}$). Sift will free the buffer space used by inactive users. Sift maintains two LRU lists: a LRU list of active users and a LRU list of recently accessed layers, that we call *UsrLRU* and *LayerLRU*, respectively. At first, Sift will select the least active user from *UsrLRU*. Next, Sift will reversely iterate *UsrLRU* until it finds a layer that is exclusively owned by that least active user, Sift then evicts the layer from layer buffer. Note that layers are shared among different active users. Sift continues with evictions until the free

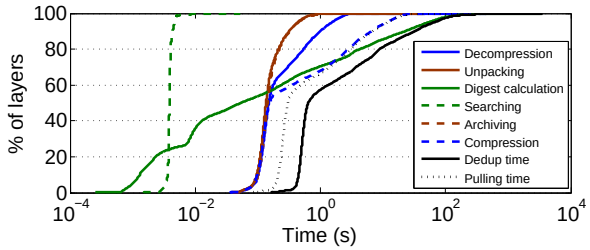


*Figure 6: Off-line file-level deduplication run time.*

space in the layer buffer is above the specified capacity threshold.

## 4 Preliminary evaluation

While Sift can effectively eliminate redundant files in the Docker registry, it introduces overhead which can reduce the registry's performance.

**Hit ratios.**

**Hit ratios with prefetching.**

**Restoring performance breakdown.**

**Simulation.** To analyze the impact of file-level deduplication on the registry performance, we conduct a preliminary simulation-based study of Sift. Our simulation approximates several of Sift's steps as described in Section 3.1. First, a layer from our dataset is copied to a RAM disk. The layer is then decompressed, unpacked, and the fingerprints of all files are computed using the MD5 hash function [31]. The simulation searches the fingerprint index for duplicates, and, if the file has not been stored previously, it records the file's fingerprint in the index. At this point our simulation does not include the latency of storing unique files. To simulate the layer reconstruction during a `pull` request, we archive and compress the corresponding files.

The simulator is implemented in 600 lines of Python code and our setup is a one-node Docker registry on a machine with 32 cores and 64 GB of RAM. To speed up the experiments and fit the required data in RAM we use 50% of all layers and exclude the ones larger than 50 MB. We process 60 layers in parallel using 60 threads. The entire simulation took 3.5 days to finish.

Figure 6 shows the CDF for each sub-operation of Sift. Unpacking, Decompression, Digest Calculation, and Searching are part of the deduplication process and together make up the Dedup time. Searching, Archiving, and Compression simulate the processing for a `pull` request and form the Pulling time.

**Push.** Sift does not directly impact the latency of `push` requests because deduplication is performed asynchronously. The appropriate performance metric for `push` is the time it takes to deduplicate a single layer. Looking at the breakdown of the deduplication time in Figure 6, we make several observations.

First, the searching time is the smallest among all operations with 90% of the searches completing in less than 4 ms and a median of 3.9 ms. Second, the calculation of digests spans a wide range from 5 $\mu$s to almost 125 s. 90% of digest calculation times are less than 27 s while 50% are less than 0.05 s. The diversity in the timing is caused by a high variety of layer sizes both in terms of storage space and file counts. Third, the run time for decompression and unpacking follows an identical distribution for around 60% of the layers and is less than 150 ms. However, after that, the times diverge and decompression times increase faster compared to unpacking times. 90% of decompressions take less than 950 ms while 90% of packing time is less than 350ms.

Overall, we see that 90% of file-level deduplication time is less than 35 s per layer, while the average processing time for a single layer is 13.5 s. This means that our single-node deployment can process about 4.4 layers/s on average (using 60 threads). In the future we will work on further improving Sift's deduplication throughput.

**Pull.** From Figure 6 we can see that 55% of the layers have close compression and archiving times ranging from from 40 ms to 150 ms and both operations contribute equally to pulling latency. After that, the times diverge and compression times increase faster with an 90$^{\text{th}}$ percentile of 8 s. This is because compression times increase for larger layers and follow the distribution of layer sizes (see Figure **??**). Compression time makes up the major portion of the pull latency and is a bottleneck. Overall, the average pull time is 2.3 s.

## 5 Discussion

We propose additional optimizations that can help to speed up Sift:

1. As the majority of the pull time is caused by compression, we propose to cache hot layers as precompressed tar files in the staging area. According to our statistics, only 10% of all images were pulled from Docker Hub more than 360 times from the time the image was first pushed to Docker Hub until May 30, 2017. Moreover, we found that 90% of pulls went to only 0.25% of images based on image pull counts. This suggests the existence of both cold and hot images and layers.
2. As deduplication provides significant storage savings, Sift can use faster but less effective local compression methods than gzip [6].
3. The registries often experience fluctuation in load with peaks and troughs [20]. Thus, file-level deduplication can be triggered when the load is low to prevent interference with client `pull` and `push` requests.

## 6 Related work

A number of studies investigated various dimensions of Docker storage performance [21, 22, 24, 34, 38, 40]. Harter et al. [24] studied 57 images from Docker Hub for a variety of metrics but not for data redundancy. Cito et al. [23] conducted an empirical study of 70,000 Dockerfiles, focusing on the image build process but not image contents. Shu et al. [33] studied the security vulnerabilities in Docker Hub images based on 356,218 images. Anwar et al. [20] performed a detailed analysis of an IBM Docker registry workload but not the dataset.

Data deduplication is a well explored and widely applied technique [26, 29, 35, 37, 42]. A number of studies characterized deduplication ratios of real-world datasets [25, 27, 28, 32, 36, 39, 41] but to the best of our knowledge, we are the first to analyze a large-scale Docker registry dataset for its deduplication properties.

## 7 Conclusion

Data deduplication has proven itself as a highly effective technique for eliminating data redundancy. In spite of being successfully applied to numerous real datasets, deduplication bypassed the promising area of Docker images. In this paper, we propose to fix this striking omission. We analyzed over 1.7 million real-world Docker image layers and identified that file-level deduplication can eliminate 96.8% of the files resulting in a capacity-wise deduplication ratio of 6.9×. We proceeded with a simulation-based evaluation of the impact of deduplication on the Docker registry performance. We found that restoring large layers from registry can slow down `pull` performance due to compression overhead. To speed up Sift, we suggested several optimizations. Our findings justify and lay way for integrating deduplication in the Docker registry.

**Future work.** In the future, we plan to investigate the effectiveness of sub-file deduplication for Docker images and to extend our analysis to more image tags rather than just the `latest` tag. We also plan to proceed with a complete implementation of Sift.

## References


[1] Android Native Development Kit (NDK). https://github.com/android-ndk/ndk.

[2] cowsay. https://github.com/piuccio/cowsay.

[3] Docker. https://www.docker.com/.

[4] Docker Hub. https://hub.docker.com/.

[5] Dockerfile. https://docs.docker.com/engine/reference/builder/.

[6] Extremely Fast Compression algorithm. https://github.com/lz4/lz4.

[7] GitHub. https://github.com/.

[8] go-ethereum. https://github.com/ethereum/go-ethereum.

[9] Google cloud storage pricing. https://cloud.google.com/storage/pricing#storage-pricing.

[10] Google test. https://github.com/google/googletest.

[11] Google's container registry. https://cloud.google.com/container-registry/.

[12] hdfs. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[13] OpenVZ Linux Containers Wiki. http://openvz.org/.

[14] python-magic. https://github.com/ahupp/python-magic.

[15] redis. https://redis.io/.

[16] singularity. http://singularity.lbl.gov/.

[17] spark. https://spark.apache.org/.

[18] Using StorReduce for cloud-based data deduplication. https://cloud.google.com/solutions/partners/storreduce-cloud-deduplication.

[19] xnu-chroot-environment. https://github.com/winocm/xnu-chroot-environment.

[20] Anwar, A., Mohamed, M., Tarasov, V., Littley, M., Rupprecht, L., Cheng, Y., Zhao, N., Skourtis, D., Warke, A. S., Ludwig, H., Hildebrand, D., and Butt, A. R. Improving docker registry design based on production workload analysis. In *USENIX FAST'18*.

[21] Bhimani, J., Yang, J., Yang, Z., Mi, N., Xu, Q., Awasthi, M., Pandurangan, R., and Balakrishnan, V. Understanding performance of I/O intensive containerized applications for NVMe SSDs. In *IEEE IPCCC'16*.

[22] Canon, R. S., and Jacobsen, D. Shifter: Containers for HPC. In *Cray User Group'16*.

[23] Cito, J., Schermann, G., Wittern, J. E., Leitner, P., Zumberi, S., and Gall, H. C. An Empirical Analysis of the Docker Container Ecosystem on GitHub. In *IEEE MSR'17*.

[24] Harter, T., Salmon, B., Liu, R., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. Slacker: Fast Distribution with Lazy Docker Containers. In *USENIX FAST'16*.

[25] Jin, K., and Miller, E. The effectiveness of deduplication on virtual machine disk images. In *ACM SYSTOR'09*.

[26] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *USENIX FAST '09*.

[27] Lu, M., Chambliss, D., Glider, J., and Constantinescu, C. Insights for data reduction in primary storage: A practical analysis. In *ACM SYSTOR'12*.

[28] Meister, D., Kaiser, J., Brinkmann, A., Cortes, T., Kuhn, M., and Kunkel, J. A study on data deduplication in HPC storage systems. In *ACM/IEEE SC'12*.

[29] Muthitacharoen, A., Chen, B., and Mazieres, D. A low-bandwidth network file system. In *ACM SOSP'01*.

[30] Pooranian, Z., Chen, K.-C., Yu, C.-M., and Conti, M. Rare: Defeating side channels based on data-deduplication in cloud storage. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (2018), IEEE, pp. 444–449.

[31] Rivest, R. The md5 message-digest algorithm.

[32] Shim, H., Shilane, P., and Hsu, W. Characterization of incremental data changes for efficient data protection. In *USENIX ATC'13*.

[33] Shu, R., Gu, X., and Enck, W. A Study of Security Vulnerabilities on Docker Hub. In *ACM CODASPY'17*.

[34] Spillane, R. P., Wang, W., Lu, L., Austruy, M., Rivera, R., and Karamanolis, C. Exo-clones: Better Container Runtime Image Management Across the Clouds. In *USENIX HotStorage'16*.

[35] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, inline data deduplication for primary storage. In *USENIX FAST'12*.

[36] SUN, Z., KUENNING, G., MANDAL, S., SHILANE, P., TARASOV, V., XIAO, N., AND ZADOK, E. A long-term user-centric analysis of deduplication patterns. In *IEEE MSST'16*.

[37] TARASOV, V., JAIN, D., KUENNING, G., MANDAL, S., PALANISAMI, K., SHILANE, P., TREHAN, S., AND ZADOK, E. Dmdedup: Device mapper target for data deduplication. In *Ottawa Linux Symposium'14*.

[38] TARASOV, V., RUPPRECHT, L., SKOURTIS, D., WARKE, A., HILDEBRAND, D., MOHAMED, M., MANDAGERE, N., LI, W., RANGASWAMI, R., AND ZHAO, M. In Search of the Ideal Storage Configuration for Docker Containers. In *IEEE AMLCS'17*.

[39] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *USENIX FAST'12*.

[40] ZHAO, F., XU, K., AND SHAIN, R. Improving Copy-on-Write Performance in Container Storage Drivers. In *SNIA SDC'16*.

[41] ZHOU, R., LIU, M., AND LI, T. Characterizing the efficiency of data deduplication for big data storage management. In *IEEE IISWC'13*.

[42] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *USENIX FAST '08*.