# 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.

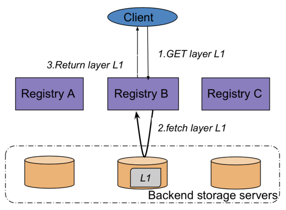

*Figure 1: PULL a layer from registry without dedup.*

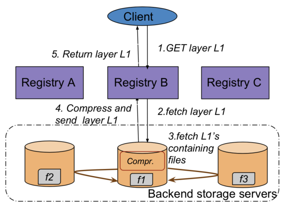

*Figure 2: PULL a layer from registry with dedup.*

## 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.

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 conducted a preliminary simulation-based study . The study aims to explore the feasibility/benefits of deduplication and quantify the overhead it introduces. ■ **Hadeel: add preliminary results**.

■ **Hadeel: 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 3 and observations in Section **??**. We present deduplication analysis in Section **??**, and Siftdesign in Sectio **??**. lastly, we describe related work in Section 8, and conclude in Section 9.

## 2 Background

■ **Hadeel: moved, will put in perfect place after finishing design** Modern container registries such as 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].

■ **Hadeel: moved, will put in perfect place after finishing design** 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.

■ **Hadeel: moved, will put in perfect place after finishing design** Intuitively, such 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. To eliminate file-level redundancy 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.

### 2.1 Docker, Docker registry, and Registry backend storage systems

Docker [3] is a popular virtualization technology that extends traditional OS containers with higher level functionalities. It allows to efficiently package an application and its runtime dependencies in a container image to simplify and automate application deployment [24].

As shown in Figure 3, a typical Docker setup consists of three main components: *client*, *host*, and *registry*. Users interact with Docker using the Docker client

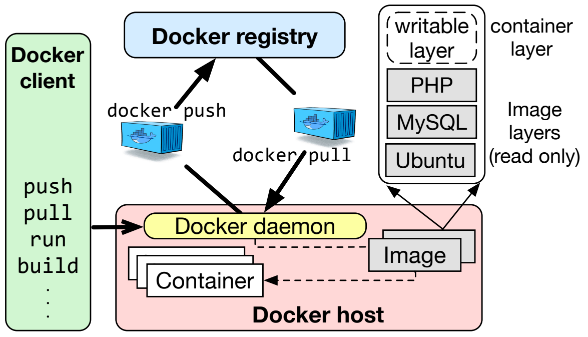

*Figure 3: Docker ecosystem.*

which, in turn, sends commands to the Docker host. The client can be co-located on the host machine. The Docker host runs a daemon process that implements the core logic of Docker and is responsible for *running* containers from locally available images. If a user tries to launch a container from an image that is not available locally, the daemon *pull*s the required image from the Docker registry. Additionally, the daemon supports *building* new images and *pushing* them to the registry.

The Docker registry 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. Image layers are stored as compressed archival files and image manifests as JSON files.

Docker Hub is one of the most popular public registries, supporting both public and private repositories, via which users can upload, search, and download images [4]. In Docker Hub, the user repositories are namespaced by user name, i.e., "$\langle username \rangle / \langle repositoryname \rangle$", while the official repositories, which are directly provided by Docker Inc. and partners are called "$\langle repositoryname \rangle$".

Modern Docker registry identifies and addresses a layer with a digest that is computed based on the uncompressed layer's content (e.g., SHA-256). Identifying layers by their content allows the registry to store only one instance of a layer even if it is referenced by multiple images. However, if at least one file differs in two otherwise identical layers, the two layers are treated as different and stored separately.

At the center of Docker is the concept of container images for packaging, distributing, and running applications. 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 (Figure 3). Any changes made to files in the image will be reflected inside the writable layer via a copy-on-write mechanism.

## 2.2 Cloud Storage and Deduplication

Most existing cloud storage providers employ data deduplication techniques to eliminate redundant data, same data stored more than once. Deduplication techniques significantly reduce storage needs and therefore reduce storage costs and improve storage efficiency. Data deduplication works by storing duplicate data chunks only once, keeping only the unique data chunks.

Current cloud providers deploy a cross-user client-side fixed-size-chunk-level data deduplication that delivers the highest deduplication gain [30]. These approaches maximize the benefit of deduplication: The cross-user data deduplication treats cloud storage as a pool shared by all the cloud users, because the potential for data deduplication is the highest as the probability for redundancies and duplicates is higher the more inclusive the shared pool. The client-side data deduplication is to ensure that only unique files are uploaded, to save network bandwidth, by having the client send a duplicate check request. The fixed-size-chunk-level specifies that a fixed-size chunk is the unit for checking for duplicates on cloud storage.

Google cloud and AWS employ StorReduce, a deduplication software that performs in-line data deduplication transparently and resides between the client's application and the hosting cloud storage. StorReduce provide 80-97% storage and bandwidth reduction to the cloud providers [18].

server-side data deduplication?

## 2.3 Web/Proxy cache

Caching is a technique widely leveraged to reduce bandwidth and load off of highly loaded bottleneck backends by temporarily storing frequently requested data. Caching can be introduced at the client, in between clients and servers as a proxy, or at the server side. Proxy caches are efficient in reducing traffic from bottleneck backends by cooperatively serving previously requested and saved data without it going all they way to the backend. The client therefore, experiences improved response times.

The Docker client inherently caches layers at the client. This way, the Docker daemon only pulls layers that are not available on the host machine.

Examples of open-source web (HTTP) proxy caches include Nginx [?], Squid [?], and Varnish [?]. Other software used for caching include Memcached [?], an open-source distributed in-memory key-value store that works as a caching system and Redis [?] which is an open-source key-value store that works as an in-memory store and as a cache.

## 2.4 Apply traditional deduplication approaches to registries?

## 2.5 Use-cases of Slimmer

**Production registries.** Google, IBM,

**Private registries.**

**Enable deduplication technique handle versatile data streams without performance degradation.**

## 3 Observations and Motivation

## 3.1 Need for Deduplication

The proliferation of Containers and Container platforms has resulted in an explosion of the number of Docker images. In order to understand the extent of storage requirements and performance demands from a Docker registry, we observe the amount of public repositories hosted by Docker Hub registry. We chose the Docker Hub registry [4] as our source of images due to its popularity and its significant number of public repositories. We noticed that the number of public repositories is constantly increasing with a growth that amounts to around 1 million repositories annually. This corresponds to 130 TB of annual growth in storage needs ■ **Hadeel: but it is actually less because of shared layers, right?**, costing around $15,000 a month if Google Cloud Storage is used [9]. This growth implies significant benefits to data deduplication.

We further performed an in-depth and empirical analysis of the viability of deduplication on container images. We analyzed around 50 TB of compressed Docker images collected from Docker Hub. In order to provide insights in terms of the redundancy measure in the image dataset and the benefits of deduplication, we have to first decompress the layers included in the images. The decompressed layer comprises a set of files. The total number of files we observed amount to over 5 billion files.

**Deduplication statistics.** A remarkable observation that emerges from the data analysis is that only around 3%■ **Hadeel: 3% or 7% ?** of the layers' constituent files are unique while the rest are redundant copies. This suggests that the current layer sharing strategy that Docker already employs is not efficient in eliminating

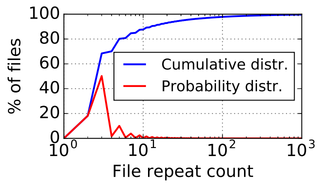

*Figure 4: File repeat count distribution.*

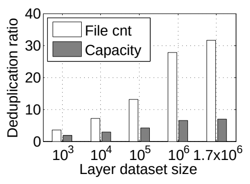

*Figure 5: Deduplication ratio growth.*

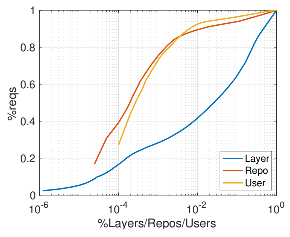

*Figure 6: Popularity of layers, repos, and users.*

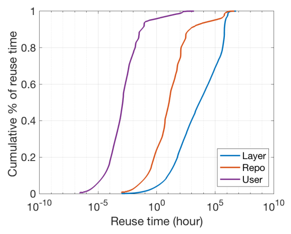

*Figure 7: CDF of reuse time for layers, repos, and users.*

duplicate data. We further analyzed the repeat count for every file and plotted the distributions as shown in Figure 4. We found that over 99.4% of files have more than one copy. Around 50% of files have exactly 4 copies and 90% of files have 10 or less copies. This indicates the high potential for file-level deduplication in the Docker registry.

**Deduplication ratio growth.** Further investigations on the potential of file-level deduplication involved analyzing the deduplication ratio. As shown in Figure 5, we analyzed the deduplication ratio and its growth for an increasing number of files stored in the registry. The x-axis values correspond to the sizes of 4 random samples drawn from the whole dataset and the size of the whole dataset.

We see that the deduplication ratio increases almost linearly with the layer dataset size. This implies that the benefits of file-level deduplication strengthens as the number of public repositories and images grow.

## 3.2 Need for User behavior based cache management

**Requests to layers are heavily skewed but layer reuse time is very long.** Figure 6 shows the registry accesses to layers, repositories, and by users for IBM container registry workload in Dallas(`dal`) [20]. There is heavy skew in layer access. For example, 25% of popular layers account for 80% of all requests. For repository accesses and accesses by users, skew is more significant than for layers. 10% most frequently accessed repositories account for 94% of all requests while 9% most active users issued 97% of all requests. Active users are the users that issue requests to registries. This means few extremely active users create their repositories in `dal` and send majority of the requests.

Figure 6 shows the reuse time of layers, repositories, and by users. Reuse time is the duration between two subsequent requests that accesses the same layer or repository while reuse time by user is defined as duration between two subsequent requests that are issued by the same user. Layer reuse time is long. The median reuse time for layer is 1.3 hour. While 80% repositories experience the highest request frequency of reuse time around 2 minutes and 90% users maintain active for at least 0.06 second. So for a registry, most of its stored layers are not accessed frequently given a very short time period while users can maintain active for a longer time. This is because users can access multiple layers and manifests.

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

## 4 Deduplication analysis

In this section, we investigate the potential for data reduction in the Docker registry by estimating the efficacy of layer sharing, compression, and the proposed file-level deduplication. We also analyze the root causes of the high file redundancy across container images.

### 4.1 Methodology

To analyze the benefits of different data reduction techniques for the Docker registry, we downloaded a large number of Docker images. We chose the Docker Hub registry [4] as our source of images due to its popularity and its significant number of public repositories. We expect that because of these reasons, our findings are applicable to other registry deployments.

Docker Hub does not provide an API to retrieve all repository names. Hence, we crawled the registry's website to obtain a list of all available repositories, and then downloaded the *latest* version of an image and its corresponding layers for each repository. We plan to extend our analysis to other image tags in the future. We downloaded 355,319 images, resulting in 1,792,609 compressed layers and 5,278,465,130 files, with a total compressed dataset size of 47 TB. To store and analyze the data, we deployed an 8-node Spark [17] cluster with HDFS [12] as the backend storage.

### 4.2 Data reduction analysis

**Layer sharing.** Compared to other existing containerization frameworks [13] [16], Docker supports the sharing of layers among different images. To analyze the effectiveness of this approach, we compute how many times each layer is referenced by images.

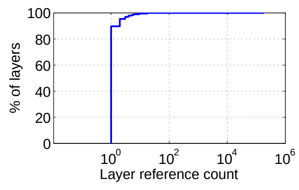

*Figure 8: CDF of layer reference count.*

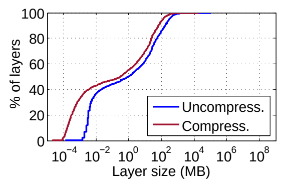

*Figure 9: CDF of compress. and uncompress. layer size.*

Figure 8 shows that around 90% of layers are referenced by a single image, an additional 5% are referenced by 2 images, and less than 1% of the layers are shared by more than 25 images. Interestingly, there is one layer that is referenced by 184,171 images. Further analysis reveals that this is an empty layer. The presence of an empty layer in an image can be explained by the fact that during the image build, Docker creates a new layer for every `RUN <cmd>` instruction in the Dockerfile [5]. If the `<cmd>`, which can be an arbitrary shell command, does not modify any files in the file system, an empty layer is created.

The next 5 top-ranked layers by reference count are included in 29,200 – 33,413 images. Specifically, one layer contains a whole `Ubuntu 14.04.2 LTS` distribution, one layer contains a `sources.list` file for `apt`, and one layer contains binaries and libraries needed for `dpkg`. The other two layers are related to cowsay, a program that can generate ASCII pictures of a cow with a message [2]. One layer contains a whole installation package for `cowsay 3.03` while the other layer only contains the binaries for cowsay.

From the above data we can estimate that without layer sharing, the Docker Hub dataset would grow from 47 TB to 85 TB, implying a **1.8**× deduplication ratio provided by layer sharing.

**Compression.** Figure 9 presents compressed and uncompressed layer size distributions. We found that 50% of the layers are smaller than 1 MB and 90% of the layers are smaller than 64 MB in compressed format. If uncompressed, 50% of the layers are smaller than 2 MB and 90% of the layers are smaller than 170 MB. Moreover, the total compressed layer dataset grows from 47 TB to 167 TB after decompression, resulting in a compression ratio of **3.6**×. Compression is complementary to layer sharing and therefore, the current registry reduces the dataset size by a factor of **3.6**×**1.8 = 6.5**.

**File-level deduplication.** Next, we calculate the deduplication ratio in terms of file count and capacity for the complete dataset. After removing redundant files, there are only 3.2% of files left that in total occupy 24 TB, resulting in deduplication ratios of **31.5**× and **6.9**× in terms of file count and capacity, respectively. With compression applied, file-level deduplication can hence reduce storage utilization by a factor of **6.9**×**3.6 = 24.8**.

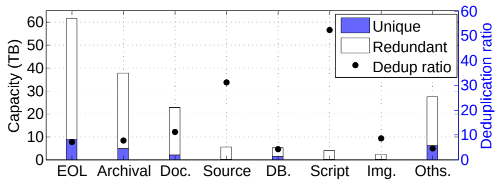

*Figure 10: Deduplication ratio for seven file classes—EOL, archival, documents, source code, database, scripts, images—and other files. Light bars indicate the original storage utilization while dark bars represent the capacity after removing redundant files. Dots shows the corresponding deduplication ratio.*

We further analyze the repeat count for every file (see Figure 4). We observe that over 99.4% of files have more than one copy. Around 50% of files have exactly 4 copies and 90% of files have 10 or less copies. The file that has the maximum repeat count of 53,654,306 is an empty file. Around 4% of empty files are `__init__.py`, which make Python treat a directory as containing packages and are usually empty. Other frequent empty files include `lock` or `.gitkeep` files.

We also analyze 5 frequently repeated files which repeat 3,338,145 – 11,847,356 times. Specifically, two files `libkrb5-3:amd64.postrm` and `libkrb5-3:amd64.postinst` are two Kerberos runtime libraries for `dpkg`. Another two files are related to the `npm` package manager (`license` and `.npmignore`) and the last file, `dependency_links.txt`, contains a list of dependency URLs for Python.

This shows that there is a high file-level redundancy in Docker images which cannot be addressed by the existing layer sharing mechanism. Hence, there is a large potential for file deduplication in the Docker registry.

**Deduplication ratio growth.** To further study the potential of file-level deduplication, we analyze the deduplication for an increasing number of files stored in the registry (see Figure 5). The x-axis values correspond to the sizes of 4 random samples drawn from the whole dataset and the size of the whole dataset.

We see that the deduplication ratio increases almost linearly with the layer dataset size. In terms of file count, it increases from **3.6**× to **31.5**× while in terms of capacity, it increases from **1.9**× to **6.9**× as the layer dataset grows from 1000 to 1.7 million layers. This confirms the high potential for file-level deduplication in large-scale Docker registry deployments.

## 4.3 Deduplication by file types

To understand the sources of high data redundancy among Docker images, we investigate deduplication for common file types. We identify 133 file types (e.g., JPG, C/C++, and Java files) using the file type library [14] and then group file types into 7 classes by their high-level use cases: 1) executable, object, and library files (EOL) (such as .o or .pyc) 2) archival (such as .gz or .tar), 3) documents (such as .txt or .tex), 4) source code (such as .c or .java), 5) scripts (such as .py or .js), 6) images (such as .png or .eps), and 7) database files (such as .sqlite or .frm). These classes cover about 88% of the whole dataset (by capacity); we group the remaining 12% in the Others class.

Figure 10 illustrates the deduplication results for these classes. The overall deduplication ratio is **6.9**× and 4 classes have a comparable ratio (indicated by the dots). For example, the deduplication ratio for EOL files is **7.1**×. However, for the other 3 classes, the deduplication ratio is significantly higher—**31.25**× for source code, **50**× for scripts, and **12.5**× for documents. This indicates that users frequently replicate source code, scripts, and documents in their images.

We found that redundant C/C++ source code takes up over 77% of the capacity occupied by source code files. Looking closer we found that, for example, Google Test [10]—a cross-platform C++ test framework available on GitHub [7]—is frequently replicated. Interestingly, we found there are multiple Docker repositories related to Google Test but there is no single *official* repository. We think that many developers replicate open source code from external public repositories, such as GitHub, and store it in container images but do not specifically encapsulate it in a separate layer. Source code also frequently changes so if different versions are put into a layer in different images, a large file base may overlap while few files are different. This can also result in different layers with high redundancy. In addition to Google Test, we also found many replicas of the source code of go-ethereum [8], Android Native Development Kit (NDK) [1], xnu-chroot-environment [19] among others.

Next, we observe that redundant EOL and archival files occupy over half of the total dataset capacity (51.4%). We analyze the top-5 most frequent archival files. We found that `NEWS.Debian.gz`, which stores news about package changes, has 521,611 copies and `pwunconv.8.gz`, which is used to create passwords has 358,374 copies. There are two files that have around 87,000 copies: `ubuntu_dists_trusty_universe_Sources.gz` and `ubuntu_dists_trusty_universe_binary-amd64_Packages.gz`, which contain metadata of the installed packages for the distributions. The last file, `gcc.log.xz` has 13,384 copies and belongs to the GNU C++ compiler.

## 5 Sift Design

In this section, we first present the challenges brought by the unique characteristics of the Docker ecosystem.

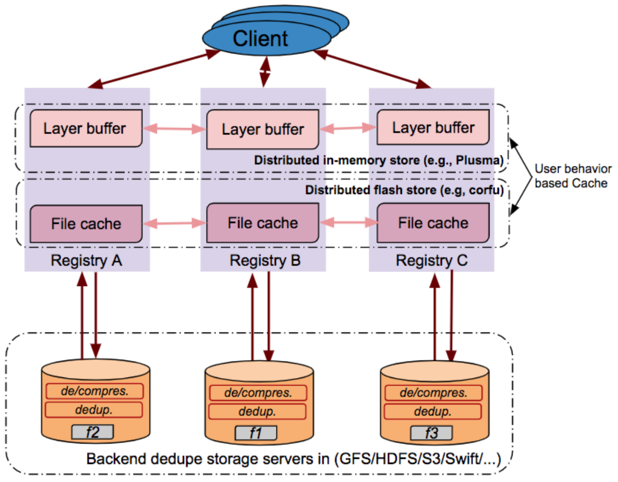
Client
Layer buffer
Layer buffer
Layer buffer
Distributed in-memory store (e.g., Plusma)
User behavior based Cache
Distributed flash store (e.g, corfu)
File cache
File cache
File cache
Registry A
Registry B
Registry C
de/compres.
dedup.
f2
f1
f3
Backend dedupe storage servers in (GFS/HDFS/S3/Swift/...)

*Figure 11: Architecture of Sift.*

Next, we discuss our design goals and finally present the design of Sift.

### 5.1 Integrating deduplication, caching, and Docker registries

**■ Ali: Following text makes no sense.. what are you trying to explain?? Sift seamlessly integrates the management of cache, deduplication on backend storage system (called backend dedup storage), and Docker registries. 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 temporary storage of frequently requested data to reduce the server's lag. They are typically deployed in a regional ISP or within a corporate network. Deduplication methods are implemented on remote backend storage servers, and transparently remove duplicates from the incoming data stream and restore the data for read requests.**

**■ Ali: Following information should be part of Background section.. Docker registry is a web server that serves Docker `pull` and Docker `push` requests. 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. 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.**

### 5.2 Design challenges

There are several 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, on average, around half of the cached layers' are not accessed again for at least 4 hours which means that if we cache a layer, we might need to wait hours to get a hit on that layer. 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.** Moreover, we have to consider that a user might deploy an applications on multiple machines, so it's not easy to predict when a user will access which layers. 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■ **Hadeel: we have to mention which trace???** analysis, only half of the `pull` 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.**

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. ■ **Ali: Explain how push layer requests are not effected?**

## 5.3 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 11). Based on our observation, user's active time is easier to predict as shown in Figure 7. 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 5.4). When a user requests a layer not present in the layer buffer, the request is forwarded to the file cache (detailed in 5.4). 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.

## 5.4 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 update 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* ■ **Hadeel: 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 5.5.

**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.

## 5.5 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 5.1, 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

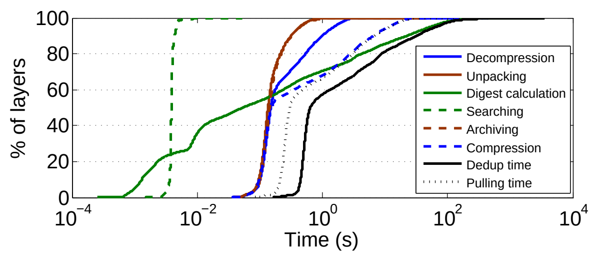

*Figure 12: Off-line file-level deduplication run time.*

buffer. Note that layers are shared among different active users. Sift continues with evictions until the free space in the layer buffer is above the specified capacity threshold.

## 6 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 5.1. 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 12 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 12, 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 12 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 9). Compression time makes up the major portion of the pull latency and is a bottleneck. Overall, the average pull time is 2.3 s.

## 7 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.

## 8 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.

## 9 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*.