Sift: A Docker Registry with Deduplication Support

Abstract

The rise of containers has led to a broad proliferation of container images. The associated storage performance and capacity requirements place high pressure on the infrastructure of registries, which store and serve images. Exploiting the high file redundancy in real-world images is a promising approach to drastically reduce the large storage requirements of the registries.

In this paper, we propose Sift, a new Docker registry architecture, which natively integrates deduplication into the registry. Sift supports several configurable *deduplication modes*, which provide different levels of storage efficiency, durability, and performance, as required for different use cases. To mitigate the negative impact of deduplication on the image download times, Sift introduces a *two-tier storage hierarchy* with a novel layer prefetch/preconstruct cache algorithm based on user access patterns. Under real workloads, for *highest data reduction mode*, Sift saves up to half of storage space compared to the current registry. For the *highest performance mode*, Sift can reduce the `GET` layer latency up to 13% compared to the state-of-the-art.

1 Introduction

Container runtimes such as Docker [**?**] leverage Linux kernel namespaces [**?**] and cgroups [**?**] to efficiently virtualize computational resources. Unlike the hardware virtualization-based Virtual Machines (VMs) [**?**], containers residing on the same host share the Operating System (OS) kernel, which yields virtual environments with lower overheads and faster starts [**?**]. Further, container runtimes introduce portable, self-contained *container images* and are surrounded by a rich ecosystem of technologies that automate and accelerate application development, deployment, and management [**?**].

Given the benefits of containers, it is not surprising that they have seen a remarkable adoption in modern cloud environments. The rapid spread of containers is further fueled by the growing popularity of microservices [**?**], for which containers act as an imperative enabler. By now, all major cloud platforms endorse containers as a core deployment technology [**?**, **?**, **?**, **?**] and their adoption is increasing. According to [**?**], in 2018, about 21% of Datadog's customers' monitored hosts run Docker and this continues to grow by about 5% annually.

Images are at the core of containerized applications. An application's container image includes the executable of the application along with a complete set of its dependencies—other executables, libraries, and configuration and data files. Images are structured in *layers*. When building an image with Docker, each executed command, such as `apt-get install`, creates a new layer on top of the previous one [**?**]. The layer contains the files that the command has modified or added during its execution. Docker leverages union file systems [**?**] to efficiently merge layers into a single file system tree when starting a container. Identical layers across different images can be shared by containers.

To store and distribute container images, Docker relies on image *registries* (e.g., Docker Hub [**?**]). Docker clients can push images to or pull them from the registries as needed. On the registry side, each layer is stored as a compressed tarball and identified by a content-based address. The Docker registry supports various storage backends for saving and retrieving layers. For example, a typical large-scale setup stores each layer as an object in an object store [**?**, **?**].

As the container market continues to expand, Docker registries have to manage a growing number of images and layers. Some conservative estimates show that in spring 2019, Docker Hub alone stored at least 2 million *public* images totaling roughly 1 PB in size [**?**, **?**]. We believe that this is just the tip of the iceberg and the number of *private* images is significantly higher. Other popular public registries [**?**, **?**, **?**, **?**], as well as on-prem registry deployments in large organizations, experience a similar surge in number of container images. As a result, organizations spend an increasing amount of their storage and networking infrastructure on operating image registries to provide an adequate level of service.

Docker images must be self-contained by definition and yet they frequently rely on common dependencies (e.g., libraries). As a result, images are prone to contain a high number of duplicate files. Docker employs layer sharing to reduce this redundancy. However, it is insufficient as layers are coarse and rarely identical due the the fact that they are built by independent developers without much coordination with others. Indeed, a recent analysis of the Docker Hub image dataset showed that about 97% of files across layers are duplicates [**?**]. Registry storage backends exacerbate the redundancy further due to the replication they perform to improve image durability and availability [**?**]. For instance, assuming 3-way replication, the number of duplicate files in the Docker Hub dataset reaches 99%.

Deduplication is an effective method to reduce capacity demands of intrinsically redundant datasets [**?**]. Docker registry, however, cannot apply deduplication to

layers directly because they come from the clients in the form of **compressed** tarballs, which do not deduplicate well. A naïve application of deduplication to the registry—decompressing layers and storing individual files in content-addressable storage—prohibitively slows down image pulls due to high layer reconstruction (layer restoring) cost. The slowdowns during image pulls are especially harmful because they contribute directly to the start times of containers. Our experiments showed that, on average, naïve deduplication increases layer download latencies by 2.5× compared to the registry without deduplication and layer reconstruction.

In this paper we propose Sift, the first Docker registry that natively supports deduplication. Sift's design is tailored to increase storage efficiency via deduplication of layers while reducing the corresponding layer restoring overhead. It employs four key techniques to reduce the impact of layer deduplication on performance:

1. Sift changes how replication is performed by the registry. It keeps an administrator-defined number of layer replicas as-is, without decompressing and deduplicating them. Accesses to these replicas do not experience layer restoring overhead. However, the additional layer replicas (needed to maintain the desired durability level) are decompressed and deduplicated.
2. Sift deduplicates less frequently accessed layers more aggressively than popular ones to speed up accesses to popular layers while achieving high storage savings.
3. By monitoring user access patterns, Sift predictively restores layers before layer download requests arrive, avoiding reconstruction latency in pull requests.
4. During layer deduplication, Sift groups files in *slices* and evenly distributes them across the cluster to parallelize and speed up layer reconstruction.

We implemented Sift as a custom layer storage system and a registry backend driver to communicate with the new backend (detailed in §5). We deployed Sift on a 14-node cluster and evaluated with real-world workloads and real layers. With the *highest performance mode*, Sift outperforms the state-of-the-art by reducing the layer downloading latency up to 13%. With the *highest deduplication mode*, Sift saves up to 50% of storage space compare to the registry without layer deduplication. The remaining deduplication modes make different trade offs in performance and data reduction (detailed in §6).

2 Docker Registries

The main purpose of a registry is to store container images and make them available to other users. A registry allows Docker clients to *push* images to and *pull* images from it. A number of public Docker registry deployments exist [**?**, **?**, **?**, **?**, **?**] and enterprises often resort to running private registries for improved security and privacy. The Docker community specifies the REST API for clients to communicate with the registries [**?**] and provides a reference registry server implementation [**?**] which serves as a basis for commercial deployments.

Docker registries group images into *repositories*, each containing versions (*tags*) of the same image, identified as `<repo-name:tag>`. For each tagged image in a repository, the Docker registry stores a *manifest*. The manifest is a JSON file, which contains the runtime configuration for a container image (e.g., environment variables) and the list of layers that make up the image. A layer is stored as a compressed archival file and identified using a digest that is computed as a hash (SHA-256) over the uncompressed content of the layer. When pulling an image, a Docker client first downloads the manifest and then the layers referenced in it (unless the layers are already present on the client). When pushing an image, a Docker client first uploads the layers (if not already present in the registry) and then uploads the manifest.

The current Docker registry software is a single-node application. To scale the registry and serve many requests concurrently, organizations typically deploy a load balancer like NGINX in front of several independent registry instances [**?**, **?**]. All the instances delegate storage and retrieval of images to drivers that interact with either a locally mounted shared file system or a remote object storage [**?**, **?**, **?**, **?**]. Upon a `PUT` layer request from a Docker client, the Docker registry receives the layer and forwards it to the backend storage driver. Typically, multiple replicas of the layer are stored for redundancy. Subsequent `GET` layer requests can be served by any registry server, which will get the layer from the backend storage using the configured driver.

3 Motivation and Challenges

In this section we first explore why file-level deduplication in Docker registries can effectively reduce storage utilization (§3.1). We then discuss the feasibility of predicting and exploiting registry access patterns for proactive layer restoration (§3.2).

3.1 Inter-layer Data Redundancy

Layers inside different container images exhibit a large amount of redundancy in terms of duplicate files. Although Docker supports the sharing of layers among different images to remove some redundant data in the Docker registry, this is not sufficient to effectively eliminate duplicates. According to the deduplication analysis of the Docker Hub dataset [**?**], only 3.2% of files are unique, resulting in a deduplication ratio of 2× in terms

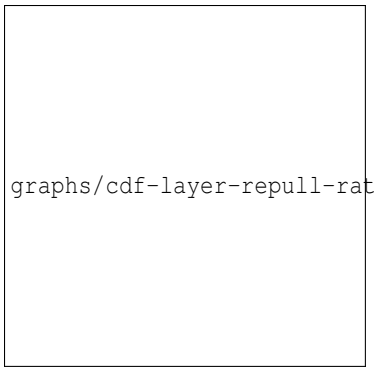


Figure 1: CDF of `GET` layer request count

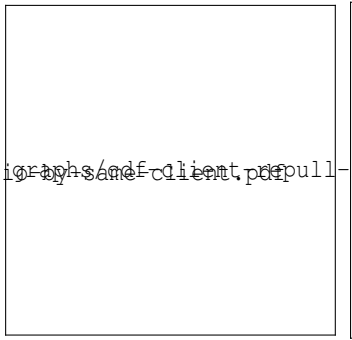


Figure 2: CDF of Client repulling probability

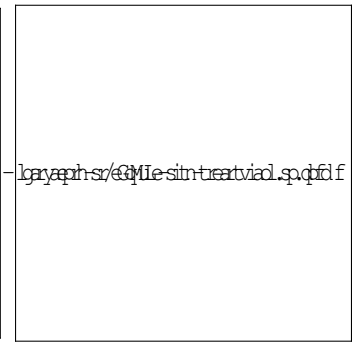


Figure 3: CDF of `GET` manifest/layer IAT

of capacity[1]. We believe that the deduplication ratio is much higher when private registries are taken into account. The duplicate files are executables, object codes, libraries, and source codes, and are likely imported by different image developers using the package installers or version control systems such as `apt`, `pip` or `git` to install similar dependencies. This file-level redundancy cannot be eliminated by the current layer-level content addressable storage system design.

As containerization frameworks like Docker keep gaining more popularity, more applications are encapsulated into images and pushed into registries. *R*-way replication for reliability additionally fuels the high storage demands of Docker registries. Hence, satisfying the demand by adding more disks and scaling out storage systems quickly becomes expensive.

File-level deduplication can address this challenge reducing the need of storage expansion. A deduplication solution needs to first decompress the compressed layer tarballs and then remove the duplicate files across different layers. However, retrieving a layer from a registry with file-level deduplication support requires restoring the layer, which involves fetching the files, archiving and compressing them[2]. These extra operations incur a considerable overhead for `GET` layer requests, which already constitute a significant portion of the container startup time [**?**]. In this paper, we explore if it is possible to deduplicate layers and save storage space without sacrificing the `GET` layer performance.

3.2 Predictable User Access Patterns

A promising approach to mitigate layer restoring overhead is predicting which layers will be accessed in the future and preconstructing them. We can exploit the fact that when a Docker client pulls an image from the registry, it will first get the manifest of the image, which includes references to the layers in the image. However, the client might or might not follow up with layer `GET` requests. Further, there might be not enough time to reconstruct the layer between the manifest and layer `GET`s.

Do clients issue repeated GETs? Typically, if a layer is already stored locally, then the client will not fetch this layer again. However, higher level container orchestrators, e.g., Kubernetes, allow users to configure different policies for starting new containers. Kubernetes, e.g., allows policies such as `IfNotPresent`, i.e, only get the layer if not present, or `AlwaysGet`, i.e. retrieve the layer, even if it is already present locally. This fact needs to be considered when predicting whether a layer will be pulled by a user or not.

We use the IBM Cloud registry workload [**?**] to analyze the likelihood for a user to *repull* an already present layer. The traces span ∼80 days for 7 different registry clusters: `Syd` (Sydney), `Dal` (Dallas), `Fra` (Frankfurt), `Lon` (London), `Dev` (Development), `Pre` (Prestaging), and `Sta` (Staging). Figure 1 shows the CDF of layer `GET` counts by the same clients. The analysis shows that the majority of layers are only fetched once by the same clients. However, there are clients, which repull the same layers continuously. For example, a client from `Lon` fetched the same layer 19,300 times.

Figure 2 shows the corresponding client repull probability, calculated as the number of repulled layers divided by the number of total `GET` layer requests issued by the same client. We see that 50% of the clients have a repull probability of less than 0.2 across all registries. We also observe that the slope of the CDFs is steep at both lower and higher probabilities but becomes flat in the middle. This suggests that we can split clients into two classes, always-pull clients and pull-once clients, e.g., based on a threshold.

[1]The deduplication ratio is 2× despite the 97% of redundant data because many of the unique files are large in size.

[2]Notice that given the large variety of Docker clients the registry-side enhancements should not involve changes on the client-side.

Is there enough time to reconstruct a layer? We analyze the duration between a `GET` manifest request and the subsequent `GET` layer request, denoted inter-arrival time (IAT). As shown in Figure 3, the majority of intervals are greater than one second. For example, 80% of intervals from `lon` are greater than one second, whereas, 60% of the intervals from `syd` are greater than five seconds.

There are several reasons for this long gap. First, when fetching an image from a registry, the Docker client fetches a fixed number of layers in parallel (three by default) starting from the lowest layers. In the case where an image contains more than three layers, the upper layers have to wait until the lower layers are downloaded. This varies the `GET` layer request arrival time for the registry. Second, network delay between clients and registry often accounts for a greater portion of the `GET` layer latency in cloud environments, especially for the large layers.

As we show later (§6) Sift can often reconstruct a layer within a single second. In the case of a small duration between a `GET` manifest request and its subsequent `GET` layer requests, layer preconstruction might not be finished in time. However, even in that case layer preconstruction can still be beneficial because the layer construction starts before the arrival of a `GET` layer request.

4 Sift Design

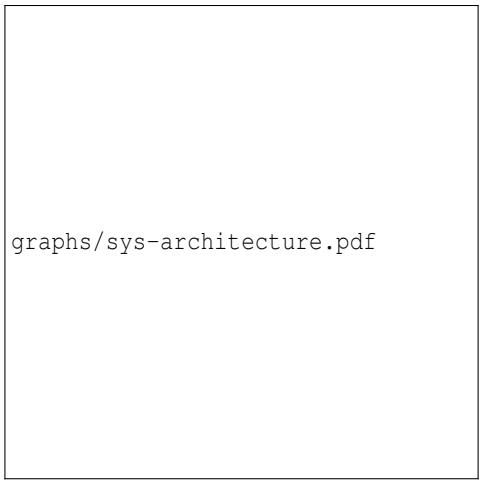
In this section, we first provide an overview of Sift (§4.1). We then describe in detail how Sift deduplicates (§4.2) and restores (§4.3) layers, and how it further improves performance via predictive cache management (§4.4).

4.1 Overview

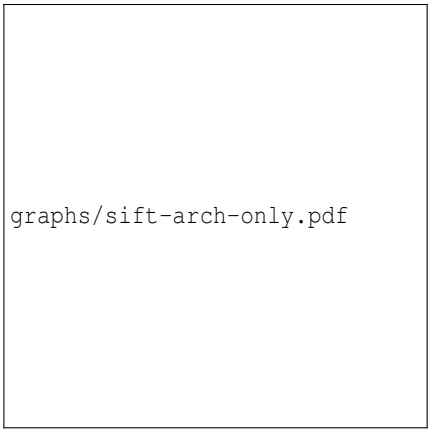
Figure 4(a) shows the architecture of Sift. Sift consists of the two main components: 1) a cluster of *storage servers*, each exposing the registry RESTful API and 2) a distributed *metadata database*. When uploading or downloading layers, Docker clients communicate with any Sift server using the registry RESTful API. Each server in the cluster contains a registry RESTful API and a backend storage system. Backend storage systems store layers and perform deduplication, keeping the deduplication metadata in the database. Sift uses three main techniques to reduce deduplication and restoring overhead: 1) replica deduplication modes; 2) parallel layer reconstruction; and 3) predictive layer prefetching/preconstruction.

Replica deduplication modes. For higher fault tolerance and availability, existing registry setups replicate each layer. Sift also performs layer replication, but in addition allows to deduplicate files inside the layers.

A *basic deduplication mode n* (B-mode *n*) defines that Sift should only keep *n* layer replicas intact and deduplicate the remaining $R-n$ layer replicas, where R is the



(a) Sift storage system



(b) Sift registry storage cluster

Figure 4: Architecture of Sift.

layer replication level. This is in comparison to R-way replication in existing registry setups that hold R layer replicas. At one extreme, B-mode R means that no replica should be deduplicated and hence, provides the best performance but no data reduction. At the other end, B-mode 0 deduplicates all layer replicas, i.e. it provides the highest deduplication ratio but adds restoration overhead for every `GET` request. The remaining B-modes in between allow to trade off performance and data reduction.

For heavily skewed workloads, Sift also provides a *selective deduplication mode* (S-mode). The S-mode utilizes the skewness in layer popularity, observed in [**?**], to decide how many replicas should be deduplicated for each layer. As there are hot layers that receive the majority of `GET` requests, S-mode sets the number of intact layer replicas proportional to its popularity. This means that hot layers have more intact layer replicas and hence, can be served faster. On the other hand, cold layers have fewer replicas to improve storage utilization.

Figure 4(b) shows an example for B-mode 1 with $R = 3$. Sift first creates 3 layer replicas across our servers. It maintains a single layer replica as the *primary layer replica* on server A for newly pushed layers. Deduplication is then carried out in one of the other servers i.e. one layer replica is decompressed and any duplicate files are discarded while unique files are kept. The unique files are replicated and saved on different servers B and C. Once deduplication is complete, the remaining two layer replicas are removed. Any subsequent `GET` layer requests are sent to server A since it stores a complete layer replica. If server A crashes, server B or C is needed to rebuild the layer and serve the `GET` request.

To support the different deduplication modes, Sift divides storage servers into two groups (Figure 4(a)): a *primary cluster* consisting of *P-servers* and a *deduplication cluster* consisting of *D-servers*. P-servers are responsible for storing full layer replicas and manifest replicas while D-servers deduplicate layer replicas at the file level, store the unique files, and replicate them. The split ensures that layer replicas and their containing file replicas are stored on different servers to maintain fault tolerance.

P- and D-servers form a 2-tier storage hierarchy. In the default case, the primary cluster serves all incoming `GET` requests. If a request can't be served from the primary cluster (e.g., due to a node failure, or Sift operating in B-mode 0 or S-modes), it will be forwarded to the deduplication cluster and the requested layer will be reconstructed.

Parallel layer reconstruction. Sift speeds up layer reconstruction through parallelism. As shown in Figure 4(a), each D-server's local storage is divided into three parts: layer stage area, preconstruction cache, and file store. The layer stage area temporarily stores newly added layer replicas. After deduplicating a replica, the resulting unique files are stored in a content addressable file store and replicated to the peer servers to provide redundancy. Once all file replicas have been stored, the layer replica is deleted from the layer stage area.

Sift distributes the layer's unique files onto several servers (see §4.2). All files on a single server belonging to the same layer are called a *slice*. A slice has a corresponding *slice recipe*, which defines the files that are part of this slice, and a *layer recipe* defines the slices needed to reconstruct the layer. This information is stored in Sift's metadata database. This allows D-servers to rebuild layer slices in parallel and thereby improve reconstruction performance. Sift maintains layer and file fingerprint indices in the metadata database.

Predictive cache prefetching. To improve the layer access latency, Sift employs a cache layer in both the primary and the deduplication cluster. Each P-server has an in-memory *user-behavior based superfetch cache* to reduce disk I/Os. When a `GET` manifest request is received from a user, Sift predicts which layers in the image will actually need to be `pulled` and *prefetches* them in the cache. To do that, Sift maintains two maps: *ILmap* and *ULmap*. ILmap stores the mapping between images and layers while ULmap keeps track of a user's access history, i.e. which layers the user has pulled and how many times (see details in §4.4).

Additionally, to avoid layer restoring overhead, each D-server maintains an on-disk layer cache (called *user-behavior based preconstruct cache*). As with the superfetch cache algorithm, when a `GET` manifest request is received, Sift predicts which layers in the image will be `pulled`, preconstructs the layers, and loads them in the preconstruct cache.

4.2 Deduplicating layers

As in the traditional Docker registry, Sift maintains a *layer index*. After receiving a `PUT` layer request, Sift first checks the layer fingerprint in the *layer index* to ensure an identical layer is not already stored. If not, Sift, replicates the layer r times across the P-servers and submits the remaining $R - r$ layer replicas to the D-servers. Those replicas are temporarily stored in the layer stage areas of the D-servers. Sift uses two consistent hashing rings to for each cluster to pick the target servers for a layer. Once the replicas have been stored successfully, Sift notifies the client of the request completion.

File-level deduplication. Following the staging area, Sift uncompresses the layer and passes the decompressed tar file to the deduplication process. Each file entry in the archive is represented as a *file header* and the associated *file content* [**?**]. The file header contains metadata such as file name, path, size, mode, owner information, etc. Sift records every file header in slice recipes (described below) to be able to correctly restore the complete layer archive later. Sift computes the file Id for each file by hashing the file content and checks, whether the file Id is already present in the file index. If the file Id is already stored, the file content will be discarded. Otherwise, it will be saved physically in the D-server's file store and the file Id is recorded in the file index.The file index maps different file Ids to the physical file replicas stored on different D-servers.

Unique file replication. Next, Sift replicates and distributes unique file replicas across D-servers. The headers and content pointers of all the files, in a given D-server, that belong to a particular layer are included in that D-server's *slice recipe* for that layer. After file replication, Sift adds the new slice recipes to the metadata database.

Sift also creates a *layer recipe* for the uploaded layer and stores it in the metadata database. The layer recipe records all the D-servers that store slices for that layer and which can act as *restoring workers*. When a layer

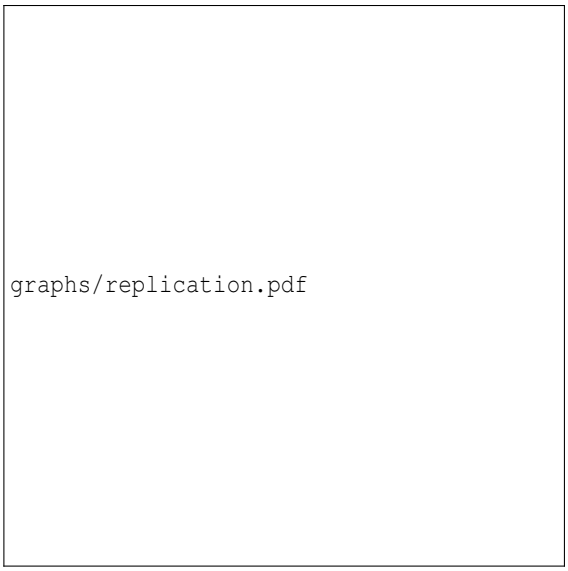


Figure 5: Layer dedup, replication, and partitioning.

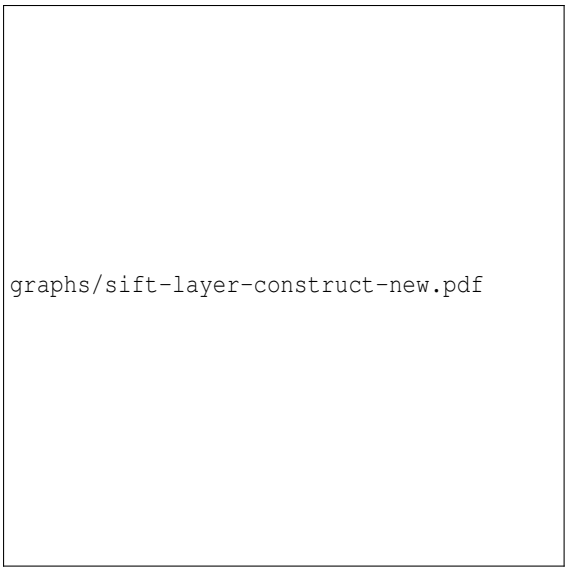


Figure 6: Parallel streaming layer construction.

needs to be reconstructed, one worker is selected as the *restoring master*, responsible for gathering all slices and rebuilding the layer (see §4.3).

Figure 5 demonstrates how Sift pipeline operates. The example assumes B-mode 1 with 3-way replication, i.e. each unique file has two replicas distributed on two different D-servers. The files $f1$, $f2$, and $f3$ are already stored in Sift, and $f1'$, $f2'$, and $f3'$ are their corresponding backup replicas. Layer $L1$ is being pushed and contains files $f1$, $f2$, $f3$, $f4$, $f5$, and $f6$. $f1$, $f2$, and $f3$ are *shared files* by $L1$ and other layers, and hence, are discarded during file-level deduplication. The unique files $f4$, $f5$ and $f6$ are added to the system and replicated to D-servers A, B, and C.

After replication, server B contains $f2$, $f5$, $f1'$, and $f4'$. Together $f2$ and $f5$ form the *primary slice* of $L1$, denoted as $L1 :: A :: P$. This slice Id contains the layer Id the slices belongs to ($L1$), the node, which stores the slice (A) and the backup level (P for primary). The two backup file replicas $f1'$ and $f4'$ on B form the *backup slice* $L1 :: B :: B$. During layer restoring, $L1$ can be restored by using any combination of primary and backup slices to achieve maximum parallelism.

Layer partitioning. To improve reconstruction times, it is important that different layer slices are equally sized and evenly distributed across D-servers. To achieve this, Sift employs a greedy packing algorithm. Consider first the simpler case in which each file only has a single replica. During layer partitioning, Sift does not migrate shared files that already have been stored on D-servers to reduce I/O overhead. Sift first computes the total size of the layer's existing shared files for each D-server (this could potentially be 0 if a D-server does not store any shared files for the layer). Next, Sift assigns the biggest unique file to the smallest partition until all the unique files are assigned.

In the case where a file has more than one replica, Sift performs the above described partitioning *per replica*. That means that it first assigns the primary replicas of the new unique files to D-servers according to the location of the primary replicas of the existing shared files. It then assigns the secondary replicas according to the location of the existing secondary replicas and so on. Additionally, Sift also ensures that two replicas of the same file are never placed on the same node.

4.3 Restoring layers

To restore a layer, each D-server in Sift uses a slice and a layer constructor. Considering Figure 5 as an illustration, the restoring process works as follows:

First, the layer constructor fetches the layer recipe from the metadata database. According to $L1$'s layer recipe, the restoring workers are D-servers A, B, and C. The node with the largest slice is picked as the restoring master (A in the example). Since A is the restoring master it sends `GET slice` requests for the primary slices to B and C. If a primary slice is missing, the master locates its corresponding backup slice and send a `GET slice` request to the corresponding D-server.

After a `GET slice` request has been received, B's and C's slice constructors start rebuilding their primary slices and send them to A as shown in Figure 6. Meanwhile A instructs its local slice constructor to restore its primary slice for $L1$. To construct a layer slice, a slice constructor

first gets the associate slice recipe from the metadata database. The recipe is keyed by a combination of layer Id, host address and requested backup level, i.e., $L1 :: A :: P$. Based on the recipe, the slice constructor creates a slice tar file by concatenating each file header and the corresponding file contents; it then compresses the slice and passes it to the master. The master concatenates all the compressed slices into a single layer compressed tarball and sends it back to the client.

The layer restoration performance is critical to keep `pull` latencies low. Hence, Sift uses several additional optimizations to further speed up the process: it parallelizes slice reconstruction on a single node and avoids generating intermediate files on disk to reduce disk I/O.

4.4 Caching and Preconstructing Layers

Sift maintains a cache layer in both the primary and deduplication clusters to speed up `pull` requests. The primary cluster cache (also called *superfetch cache*) is memory-based to avoid disk I/O during layer retrievals while the deduplication cluster cache (also called *preconstruct cache*) is on disk. The main purpose of the preconstruct cache is to store preconstructed layers, which are likely to be accessed in the future, to avoid the layer restoring overhead. Both caches are filled based on the user access patterns observed in §3.

Request prediction. To accurately predict layers that will be accessed in the future, Sift keeps track of image metadata and user access patterns in two data structures: *ILmap* and *ULmap*. ILmap maps an image to its containing *layer set*. ULmap stores, for each user, the layers the user has accessed and the corresponding pull count. A user is uniquely identified by extracting the sender network address from the request.

When a `GET manifest` request r is received, Sift first calculates a set of image layers that haven't been pulled by the user $r.addr$ by calculating the difference S_Δ between the image's layer set and the user's accessed layer set:

$$S_\Delta = ILmap[r.img] - ULmap[r.addr].$$

The layers in S_Δ are likely to be accessed soon.

Recall from §3.2 that some users *always* pull layers, no matter if the layers have been previously pulled. To detect such users, Sift first computes the subset, $S_\cap$, of layers from the image that have already been pulled by the user by computing

$$S_\cap = ILmap[r.img] \cap ULmap[r.addr].$$

Next, Sift compares the client repull probability $\gamma[r.addr]$ with a predefined client repull threshold ε. If $\gamma[r.addr] > \varepsilon$, then Sift classifies the user as a repull user and fetches the layers in $S_\cap$ into the cache.

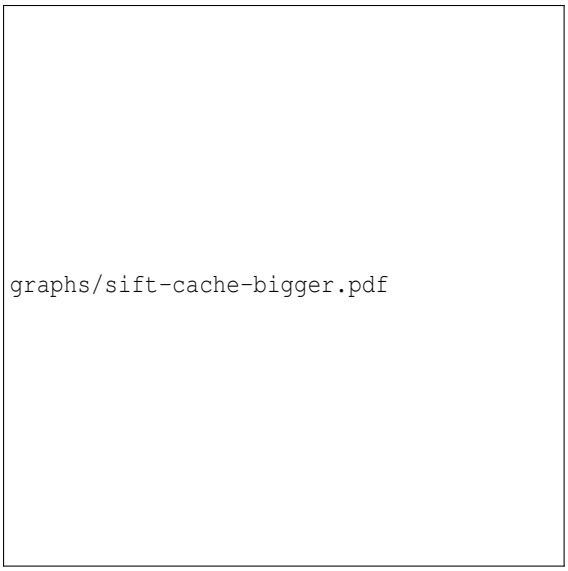


Figure 7: Tiered storage architecture.

The repull probability is computed using ULmap. For each `GET manifest` request r, Sift will compute the repull probability for a user $r.addr$ as

$$\gamma[r.addr] = \sum_{l \in RL} l.pullCount / \sum_{l \in L} l.pullCount$$

where RL is the set of layers that the user has repulled before (i.e., with a pull count greater than 1) and L is the set of all layers the user has ever pulled. Sift updates the pull counts every time it receives a `GET layer` request.

Cache handling in tiered storage. The introduction of the two cache layers results in a 5-level 2-tier storage architecture of Sift as shown in Figure 7. Requests are passed through the tiers in the order displayed in the figure. Upon a `GET` layer request, Sift first determines the P-server(s) which is (are) responsible for the layer and searches the superfetch cache(s). If the layer is present the request will be served from cache. Otherwise, the request will be served from the layer store .

If a `GET layer` request cannot be served from the primary cluster due to a failure of the corresponding P-server(s), the request will be forwarded to the deduplication cluster. In that case, Sift will first look up the layer recipe. If not found, it means that the layer has not been fully deduplicated yet and Sift will serve the layer from one of the layer stage areas of the responsible D-servers. Those are located through consistent hashing. If the layer recipe is present, Sift will contact the restoring master to check, whether the layer is in its preconstruct cache. Otherwise, it will instruct the restoring master to rebuild the layer.

Both the superfetch and the preconstruct caches are write-through caches. When a layer is evicted, it is simply

discarded since the layers are read-only. We use an Adaptive Replacement Cache (ARC) replacement policy [**?**], which keeps track of both the frequently and recently used layers and adapts to changing access patterns.

5 Implementation

We implement Sift in Go on top of the existing Docker registry implementation. In our implementation, we address a set of challenges regarding distributed, reliable handling of metadata (§5.1), dealing with non-regular files (§5.2), and generating a suitable workload (§5.3).

5.1 Handling metadata

Sift's metadata database needs to be reliable and fault tolerant to always be able to correctly restore deduplicated layers in case of a failure. Sift uses Redis [**?**] to store its metadata, i.e. the slice and layer recipes, the file and layer indices, and the ULmap and ILmap. We enable *append-only file (AOF)* to log all changes for durability purposes. This allows us to rebuild the dataset and prevent data loss after a Redis restart. Moreover, we configure Redis to save RDB (Redis Database File) snapshots every few minutes for additional reliability. To improve availability, we configure Redis to use 3-way replication.

To ensure that the metadata is in a consistent state, Sift uses Redis as a distributed lock to make sure that no file duplicates are stored in registry cluster. For the file and layer indices and the slice and layer recipes, each key can be set to hold its value only if the key does not yet exist in Redis (i.e, using `SETNX` [**?**]). When a key already holds a value, a file duplicate or layer duplicate is identified and is removed from the registry cluster.

Additionally, Sift maintains a synchronization map to ensure that multiple layer restoring processes don't attempt to restore the same layer simultaneously. If a layer is currently being restored, subsequent `GET` layer requests to this layer wait until the layer is restored. Other layers, however, can be constructed in parallel. In this case, redundant layer restoring is avoided, saving I/O, network, and computation resources.

5.2 Deduplication details

While deduplicating layers, Sift needs to be able to deal with non-regular files such as symlinks, device files, or directories. Therefore, Sift only considers regular files for deduplication. After removing regular file duplicates, any remaining unique files are saved in a local file store.

/var/lib/registry/docker/registry/v2 is the default root directory in the local file system, where all registry-related data is stored (denoted as *rootdir*). On the deduplication cluster, each uploaded layer is stored in directory *rootdir/blobs/sha256/ld.2/ld/* as a file named `data` where ld is the layer's hex digest and ld.2 are the first two digits of the hex digest. When deduplication has finished, `data` is deleted.

During deduplication, all *regular* unique files are distributed to their corresponding D-servers and stored there locally under *rootdir/uniquefiles/sha256/fd.2/fd/*, where fd is the file's hex digest and fd.2 its first two digits. Every *non-regular* file will be stored on the D-server which initially received the layer for deduplication, under *rootdir/blobs/sha256/ld.2/ld/tmp*. When a layer is reconstructed, the restoring master can check for non-regular files through the layer digest under that location.

5.3 Workload generation

To test Sift's ability to deduplicate, we need a representative, production registry workload. We base our workload generator on the IBM cloud registry trace replayer [**?**]. However, as the traces do not contain specific image data and the trace replayer only generates random layers, we modify the replayer to match requested layers in the IBM cloud registry trace with real layers, downloaded from Docker Hub. Specifically, we split the downloaded layers into different groups of similar size. We then extract layer digests from layer requests recorded in the IBM trace and randomly match each digest to a layer in a specific group. This allows us to replay a workload and control the layer size. Consequently, each layer request pulls or pushes a real layer. For manifest only requests, we generate a random file to emulate the manifest file.

In addition, our workload generator uses a proxy emulator to decide the server for each request. The proxy emulator uses consistent hashing [**?**] to distribute layers and manifests. It maintains a ring of registry servers and calculates a destination registry server for each `push` layer or manifest request by hashing its digest. For `pull` manifest requests, the proxy emulator maintains two consistent hashing rings, one for the P-servers, and one for the D-servers. By default, it first queries the P-servers but if the requested P-server is not available, it pulls from the D-servers.

6 Evaluation

For the evaluation of Sift, we are interested in two main aspects. First, the deduplication performance of Sift happening in the D-cluster (§6.2). Second, the effect of different deduplication modes on the performance of the primary cluster (§6.3).

Table 1: Workload parameters.

Trace	#GET L	#GET M	#PUT L	#PUT M	#Uniq L	#Uniq M
Dal	2867	2000	124	9	1278	88
Fra	1602	3278	111	9	420	43
Lon	924	3972	98	6	698	88
Syd	1310	3653	35	2	154	18

6.1 Evaluation Setup

Our testbed consists of a 16-node cluster with each node being equipped with 8 cores, 16 GB RAM, a 500 GB SSD, and a 10 Gbps NIC.

Dataset. As our dataset, we download 74,000 popular images (i.e., images with a pull count greater than 100) from Docker Hub. The total size of this dataset is 12.5 TB and contains 507,023 layers with 87.5% of them being smaller than 50 MB. After decompressing the layers, the total size of the dataset is 27.7 TB. Applying file-level deduplication on the decompressed dataset reduces the total size of the dataset to only 7.2 TB, yielding a deduplication ratio of 1.74.

Workload. To evaluate how Sift performs for production registry workloads, we use the IBM Cloud Registry traces [**?**]. The traces come from three private registry clusters, four production registry clusters and span approximately 80 days. As detailed in §5, we randomly match layers from the four anonymized production traces (`dal`, `fra`, `lon`, and `syd`) to layers of our dataset to generate four real-world production workloads. Before each experiment, we preload all unique layers to the P- and D-servers and then replay the first 5,000 requests from each workload. Tables 1 details how the 5,000 requests for each workload are composed with respect to layer and manifest requests.

6.2 Deduplication Performance

In this section, we evaluate the layer restoring latency of Sift's deduplication cluster and its impact on `GET` layer request latency. We first measure a single D-server's layer restoring capability and compare it with a *no deduplication* scheme, i.e. a plain registry server with a local file system backend that performs no deduplication. Then we scale out to a deduplication cluster with multiple D-servers and compare Sift to BOLT [**?**], a recently published, state-of-the-art registry design.

Restoring latency. To evaluate Sift's restoring latency, we launch a single node registry on a server and use one client on a separate machine. We compare four different configurations: (1) no deduplication; (2) Sift without caching; (3) Sift with ARC caching but no preconstruction; and (4) Sift with preconstruct cache. For each setup, we use the local file system as the storage backend and replay the `dal` workload with layer size groups from 1 MB to 9 MB. For this experiment, we focus only on smaller layers as our registry setup is single-node.

As shown in Figure 8, layer restoring increases the average `GET` layer request latency by 189% for layers with size 1 MB for Sift without caching compared to *no deduplication*. Moreover, the layer restoring latency increases almost linearly as the layer size increases. While a 1 MB layer takes 0.13 s to restore and download, it takes 0.22 s for a 9 MB layer for Sift without caching.

The results show that leveraging a cache can largely reduce layer restoring latency. When using an ARC cache, the layer restoring overhead decreases by 40% for layers with a size of 1 MB compared to Sift without cache. However, as the layer size increases, the improvement drops. For 9 MB layers, the presence of the ARC cache reduces the average layer restoring overhead by only 16% from the overhead of using Sift without caching. With the preconstruct cache, Sift is able to improve `GET` layer latencies even further. For 1 MB layers, the average latency decreases by an additional 24% compared to the ARC cache. Overall, Sift with preconstruct incurs the lowest overhead and only increases latencies by 19% for layers with a 9 MB size.

Cache hit ratio. Figure 9 shows the cache hit ratios for the ARC and preconstruct caches. Note that the cache size is set to 20% of `dal`'s ingress data ,i.e. 20% of the total size of all unique layers, which are pushed to the registry as part of our workload. The cache hit ratio for ARC is stable at 0.77 for all layer sizes while the preconstruct cache achieves a hit ratio of 0.95 for 1 MB.

As the layer size increases to 9 MB, the preconstruct cache hit ratio decreases to 79%. This is because the layer restoring latency increases with layer size as shown in Figure 8 and therefore, some layers can not be preconstructed on time. This is indicated by the increasing number of waiting `GET` layer requests for larger layer sizes, depicted by the preconstruct cache waiting ratio in Figure 9. The waiting ratio is the number of `GET` requests that were blocked on reconstruction divided by the total number of `GET` requests. The user-behavior based request prediction accuracy is calculated by summing the preconstruct cache hit ratio and the waiting ratio, i.e. all requests that successfully retrieved layers from the cache, which results to 0.95.

Inter-arrival time of manifest and layer requests. Next, we vary the inter-arrival time (IAT) between a `GET` manifest request and its subsequent `GET` layer requests. The layer size used here is 9 MB. Figure 10 compares the average `GET` layer latency for Sift with preconstruct cache versus no deduplication.

When the IAT is 1 s, Sift with preconstruct cache only imposes a 19% overhead. However, as the IAT decreases, the average `GET` layer latency increases because layers are not preconstructed in time. When the client replays

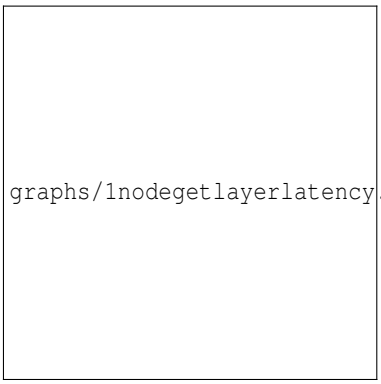


Figure 8: GET layer latency

Figure 9: Cache hit ratio

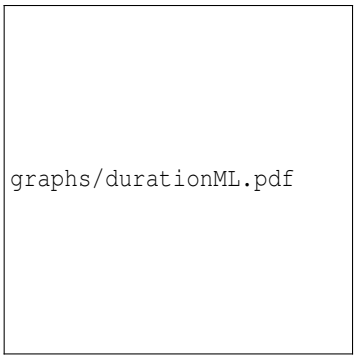


Figure 10: `GET` manifest/layer IAT

Figure 11: GET layer latency with different cluster size

Figure 12: GET layer latency with different client concurrency

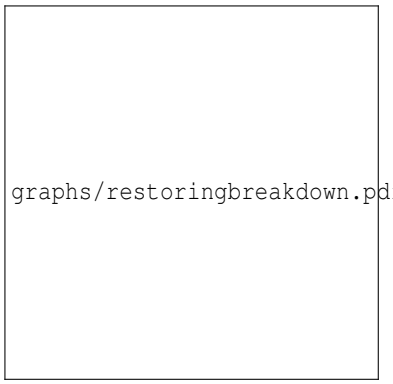


Figure 13: Restoring latency breakdown

requests as fast as possible, the overhead of restoring layers on `GET` layer requests increases by 30%.

Cluster scale out. We analyze the impact of larger clusters and layers by increasing the number of registry servers from 7 to 14, and the layer sizes from 30 to 70 MB and compare Sift to the BOLT registry. Each registry server uses the local file system to store, and in the case of Sift, deduplicate layers. Similarly to BOLT, the consistent hashing logic is implemented at the client side (see §5) to route requests to the correct registry server. We launch 20 clients on 2 servers to replay requests to the registries.

As shown in Figure 11, BOLT achieves stable request latencies when the number of registry servers increases. While Sift performs worse than BOLT, its performance improves with a larger cluster size. This is because with a bigger deduplication cluster, layers can be restored faster due to higher parallelism. For example, for a layer size of 30 MB, doubling the cluster size reduces the restoring overhead by 35%.

Workload scale up. Next, we evaluate Sift's performance for an increasing workload. We vary the number of concurrent client requests and measure the average `GET` layer request latency on a 14-node cluster for Sift and BOLT. The results are shown in Figure 12.

BOLT only experiences a slight increase in `GET` layer latency as the number of clients increase. On the other hand, latencies for Sift increase linearly with the number of concurrent client requests. For example, for 20 concurrent requests, the average `GET` layer latency is 0.37 s which increases to 0.47 s for 60 concurrent requests. This is because layer restoring is computationally intensive, i.e. for more concurrent requests, the CPU becomes a bottleneck.

Restoring latency breakdown. To analyze Sift's performance in more detail, we measure the time it takes to reconstruct a layer and break the process down into its individual steps. The steps in layer reconstruction include looking up the layer recipe, fetching and merging slices, and transferring the layer. Fetching and merging slices in itself involves slice recipe lookup, slice construction, and slice transfer. The latencies are measured on the 7-node cluster and the breakdown is shown in Figure 13.

We can see that slice reconstruction accounts for the

Figure 14: Deduplication latency breakdown

Figure 15: GET layer latency for different layer sizes

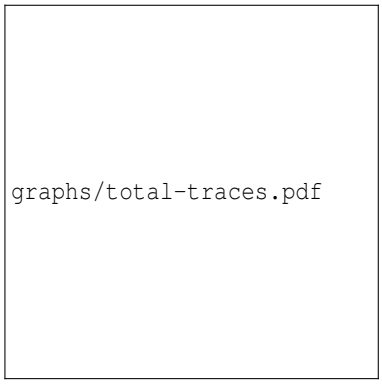


Figure 16: GET layer latency for different workloads

largest portion of layer construction time. Slice construction involves file archiving and compression, which are the main bottlenecks. Layer construction time increases with layer size because layer slices become bigger and take longer to archive and compress. For example, for layers of size 30 MB, fetching and merging slices takes 1.3 s on average while for 70 MB layers, it takes 3.6 s.

Deduplication latency breakdown. Finally, we study the deduplication process in more detail. Note that to reduce the impact of layer deduplication on the GET layer request, the layer deduplication process runs in off-line mode. Figure 14 shows the breakdown of the deduplication latency.

We observe that decompression and file-level deduplication account for the largest portions of the layer deduplication duration. This is because Sift uses single-threaded decompression and file-level deduplication in order to reduce interference with the critical layer restoration processes. Moreover, the duration of decompression and file-level deduplication increases with layer sizes. For example, when the layer size increases from 30 MB to 70 MB, the decompression duration increases from 1.6 s to 3.8 s on average and the file-level deduplication duration increases from 2.5 s to 8.8 s on average. This is expected as more data needs to be processed.

6.3 Deduplication Modes

Next, we evaluate the performance of Sift's primary cluster and different B-mode configurations. We test five modes of Sift on a 9-node cluster: B-mode 0, 1, 2, and 3, and S-mode. B-mode 0 deduplicates *all* layer replicas while B-mode 3 does not perform deduplication. Accordingly, B-mode 0 doesn't have P-servers while B-mode 3 doesn't have D-servers. For the remaining two B-modes, B-mode 1 and B-mode 2, the number of P-servers is set to 3 and 6, respectively. For S-mode, the number of P-servers is set to 6. We use 100 clients to replay the traces and BOLT with 3-way replication on the same 9-node setup as our baseline.

Different layer sizes. Figure 15 shows the average GET layer response for the dal workload, varying the layer size (from 5 MB to 25 MB). Note that for B-mode 0, the GET layer latency refers to the layer restoring latency as no complete layer replica is stored.

As expected, the response times increase with a larger layer size for all configurations. Overall, B-mode 3 achieves the lowest latencies as it is not performing any deduplication. Compared to BOLT, B-mode 3 benefits from Sift's superfetch cache, which lowers latencies by prefetching layers into and serving them from memory. B-mode 3 is able to reduce the average GET latencies by 13% compared to BOLT for a layer size of 20 MB. The superfetch cache hit ratio in this case is 0.91.

Looking at B-mode 0, we see that it adds the highest overhead, up to 37%, to the GET layer latencies as it deduplicates everything and maximizes storage savings. The remaining B-modes are in between with B-mode 1 showing slightly higher latencies compared to B-mode 2 as B-mode 1 only keeps one complete layer replica. Additionally, B-mode 1 only has 3 P-servers, which intensifies the load on the primary cluster. While the difference is minor for layer sizes less than or equal to 20 MB, only up to 18%, it is around 30% for 25 MB layers.

The GET layer performance of S-mode is similar to that of *B-mode 2*, e.g., the average GET layer latency variation between the two modes is only 3% when the layer size is 20 MB. This is because in S-mode, the popular layers have multiple replicas across the 6-node cluster, which improves the overall performance. Additionally, S-mode can reduce storage consumption by 40% compared to B-mode 2 as less popular layers keep less complete replicas.

Different traces. Next, we replay all four traces from the IBM registry deployments with a layer size of $\sim$25 MB

for all the requests. Figure 16 shows the average GET layer latency across the four different traces.

Overall, we observe a similar trend to the previous analysis of the dal trace: B-mode 3 shows the lowest overhead, followed by B-modes 2, 1, and 0 (in that order) and BOLT performs similar to B-mode 2 while S-mode is worse than B-mode 2 but better than B-modes 1 and 0.

The lon trace has the lowest GET layer latency due to the fact that lon has fewer GET layer requests compared to other traces as shown in Table 1. The majority of requests in lon are GET manifest requests, which are not affected by Sift's deduplication.

Figure 16 illustrates that due to layer restoration overhead B-mode 0 has a 39% and 80% higher GET layer latency compared to BOLT for fra and lon, respectively. We also observe that fra and lon have lower preconstruct cache hit ratios of 0.72 and 0.8, respectively. In turn, the fra and lon traces have 21% and 15% of GET layer requests waiting for layer construction, respectively. because of the shorter inter-arrival time between GET manifest requests and their subsequent GET layer requests in those traces. Note that the *superfetch cache* hit ratio is not as sensitive to inter-arrival times as it does not need to wait for a layer to be preconstructed before it can be loaded into the cache. Therefore, *B-mode 3* outperforms *BOLT* for both the fra and lon traces.

The difference between the modes in syd is less pronounced as the inter-arrival times between GET manifest and GET layer requests is higher on average and hence, the cache hit ratio increases to 0.96 and most layers are served from the cache in all modes.

7 Related Work

A number of studies investigated various dimensions of Docker storage performance [**?**, **?**, **?**, **?**, **?**, **?**, **?**]. However, none of them provide deduplication capabilities for the registry.

Skourtis et al. [**?**] proposed to reduce registry storage utilization by restructuring layers to maximize their overlap. However, they change the existing structure of images whereas Sift leaves images unchanged. There is one community proposal to add file-level deduplication to container images [**?**], but doesn't provide a detailed design or performance analysis. There is other work aimed at reducing image sizes [**?**, **?**, **?**, **?**] which are orthogonal to our approach and can be used with Sift.

Deduplication in cloud storage has been investigated for decades, particularly for virtual machine images [**?**, **?**, **?**, **?**]. Many studies also focused on primary and backup data deduplication [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**] and show the effectiveness of file- and sub-file-level deduplication [**?**, **?**]. Sift utilizes file-level deduplication but is specifically designed for Docker registries, which allows it to leverage image and workload information to reduce deduplication overhead.

8 Conclusion

We presented Sift, a new Docker registry architecture that supports deduplication for Docker images to reduce storage utilization. Sift makes several design decisions to reduce deduplication overhead and keep layer retrieval latencies low. It parallelizes layer reconstruction locally and across the cluster to mitigate the overhead of compression and decompression of layers, and exploits user behavior patterns to preconstruct layers before being accessed. Sift additionally leverages caching at different storage tiers to further improve GET latencies. Our evaluation shows that we can achieve a deduplication ratio of 1.74× while only adding an overhead of up to 80% when layers are pulled.

References

[1] Aliyun Open Storage Service (Aliyun OSS). https://cn.aliyun.com/product/oss?spm=5176.683009.2.4.Wma3SL.

[2] cgroups - Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html.

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

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

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

[6] Microsoft azure storage. https://azure.microsoft.com/en-us/services/storage/.

[7] namespaces: namespaces - overview of Linux namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html.

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

[9] K. Adams and O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. *ACM SIGOPS Operating Systems Review*, 40(5):2–13, 2006.

[10] Alfred Krohmer. Proposal: Deduplicated storage and transfer of container images. https://gist.github.com/devkid/5249ea4c88aab4c7bff1b34c955c1980.

[11] Amazon. Amazon elastic container registry. https://aws.amazon.com/ecr/.

[12] Amazon. Containers on aws. https://aws.amazon.com/containers/services/.

[13] A. Anwar, M. Mohamed, V. Tarasov, M. Littley, L. Rupprecht, Y. Cheng, N. Zhao, D. Skourtis, A. S. Warke, H. Ludwig, D. Hildebrand, and A. R. Butt. Improving Docker Registry Design Based on Production Workload Analysis. In *16th USENIX Conference on File and Storage Technologies (FAST)*, 2018.

[14] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-organized, Fault-tolerant and Scalable Replication Scheme for Cloud Storage. In *1st ACM Symposium on Cloud Computing (SoCC)*, 2010.

[15] R. S. Canon and D. Jacobsen. Shifter: Containers for HPC. In *Cray User Group*, 2016.

[16] Cloud Native Computing Foundation Projects. https://www.cncf.io/projects/.

[17] Datadog. 8 Surprising Facts about Real Docker Adoption. https://www.datadoghq.com/docker-adoption/.

[18] Docker. Docker Registry. https://github.com/docker/distribution.

[19] Docker. Docker Registry HTTP API V2. https://github.com/docker/distribution/blob/master/docs/spec/api.md.

[20] DockerSlim. https://dockersl.im.

[21] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *USENIX Annual Technical Conference (ATC)*, 2014.

[22] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[23] Y. Fu, H. Jiang, N. Xiao, L. Tian, and F. Liu. AA-Dedupe: An Application-aware Source Deduplication Approach for Cloud Backup Services in the Personal Computing Environment. In *IEEE International Conference on Cluster Computing (Cluster)*, 2011.

[24] GNU Tar. Basic Tar Format. https://www.gnu.org/software/tar/manual/html_node/Standard.html.

[25] Google. Google container registry. https://cloud.google.com/container-registry/.

[26] Google compute engine. Google Compute Engine. https://cloud.google.com/compute/.

[27] K. Gschwind, C. Adam, S. Duri, S. Nadgowda, and M. Vukovic. Optimizing Service Delivery with Minimal Runtimes. In *International Conference on Service-Oriented Computing (ICSOC)*, 2017.

[28] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Slacker: Fast Distribution with Lazy Docker Containers. In *14th USENIX Conference on File and Storage Technologies (FAST)*, 2016.

[29] IBM Cloud Kubernetes Service. Ibm cloud kubernetes service. https://www.ibm.com/cloud/container-service.

[30] IBM Cloud Kubernetes Service. S3 storage driver. https://docs.docker.com/registry/storage-drivers/s3/.

[31] K. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An Empirical Analysis of Similarity in Virtual Machine Images. In *Middleware Industry Track Workshop*, 2011.

[32] JFrog Artifcatory. https://jfrog.com/artifactory/.

[33] K. Jin and E. L. Miller. The Effectiveness of Deduplication on Virtual Machine Disk Images. In *International Systems and Storage Conference (SYSTOR)*, 2009.

[34] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing (STOC)*, 1997.

[35] K. Kumar and M. Kurhekar. Economically Efficient Virtualization over Cloud Using Docker Containers. In *IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2016.

[36] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving Restore Speed for Backup Systems that use Inline Chunk-based Deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[37] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse Indexing:

Large Scale, Inline Deduplication Using Sampling and Locality. In *7th USENIX Conference on File and Storage Technologies (FAST)*, 2009.

[38] M. Littley, A. Anwar, H. Fayyaz, Z. Fayyaz, V. Tarasov, L. Rupprecht, D. Skourtis, M. Mohamed, H. Ludwig, Y. Cheng, and A. R. Butt. Bolt: Towards a Scalable Docker Registry via Hyperconvergence. In *IEEE International Conference on Cloud Computing (CLOUD)*, 2019.

[39] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for Data Reduction in Primary Storage: A Practical Analysis. In *International Systems and Storage Conference (SYSTOR)*, 2012.

[40] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[41] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A Study on Data Deduplication in HPC Storage Systems. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[42] Microsoft. Azure container registry. https://azure.microsoft.com/en-us/services/container-registry/.

[43] Microsoft Azure. https://azure.microsoft.com/en-us/.

[44] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *ACM SIGOPS Operating Systems Review*, volume 35, 2001.

[45] OpenStack Swift storage driver. Openstack swift storage driver. https://docs.docker.com/registry/storage-drivers/swift/.

[46] J. Paulo and J. Pereira. A survey and classification of storage deduplication systems. *ACM Computing Surveys (CSUR)*, 47(1):11, 2014.

[47] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: Automatically Debloating Containers. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017.

[48] Redis. SETNX. https://redis.io/commands/setnx.

[49] H. Shim, P. Shilane, and W. Hsu. Characterization of Incremental Data Changes for Efficient Data Protection. In *USENIX Annual Technical Conference (ATC)*, 2013.

[50] D. Skourtis, L. Rupprecht, V. Tarasov, and N. Megiddo. Carving Perfect Layers out of Docker Images. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2019.

[51] R. P. Spillane, W. Wang, L. Lu, M. Austruy, R. Rivera, and C. Karamanolis. Exo-clones: Better Container Runtime Image Management Across the Clouds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2016.

[52] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[53] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok. A Long-Term User-Centric Analysis of Deduplication Patterns. In *32nd International Conference on Massive Storage Systems and Technology (MSST)*, 2016.

[54] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok. Dmdedup: Device Mapper Target for Data Deduplication. In *Ottawa Linux Symposium*, 2014.

[55] V. Tarasov, L. Rupprecht, D. Skourtis, A. Warke, D. Hildebrand, M. Mohamed, N. Mandagere, W. Li, R. Rangaswami, and M. Zhao. In Search of the Ideal Storage Configuration for Docker Containers. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2017.

[56] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci. Cntr: Lightweight OS Containers. In *USENIX Annual Technical Conference (ATC)*, 2018.

[57] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu. Characteristics of Backup Workloads in Production Systems. In *10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[58] E. Wolff. *Microservices: Flexible Software Architecture*. Addison-Wesley Professional, 2016.

[59] F. Zhao, K. Xu, and R. Shain. Improving Copy-on-Write Performance in Container Storage Drivers. In *Storage Developer Conference (SDC)*, 2016.

[60] N. Zhao, V. Tarasov, H. Albahar, A. Anwar, L. Rupprecht, D. Skourtis, A. S. Warke, M. Mohamed, and A. R. Butt. Large-scale analysis of the docker hub dataset. In *IEEE International Conference on Cluster Computing (Cluster)*, 2019.

[61] R. Zhou, M. Liu, and T. Li. Characterizing the efficiency of data deduplication for big data storage management. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2013.

[62] B. Zhu, K. Li, and R. H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *6th USENIX Conference on File and Storage Technologies (FAST)*, 2008.