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

The fast growing number of container images and the associated performance and storage capacity demands are key obstacles to sustaining and scaling Docker registries. The inability to efficiently and effectively deduplicate Docker image layers that are stored in a compressed format is a major obstacle. In this paper, we propose a new Docker registry architecture, Sift, that integrates caching and deduplication with Docker registries to increase deduplication efficacy and reduce storage needs while mitigating any performance overhead. Sift uses a highly-effective user-access-history-based prefetch algorithm, and a two-tier heterogeneous cache comprising memory and flash storage. The approach enables it to achieve a 96% hit ratio and save 56% more cache space.

1 Introduction

Containers have become an effective means for deploying modern applications because of desirable features such as tight isolation, low overhead, and efficient packaging of the execution environment [3]. However, sustaining and scaling container systems in the face of exponential growth is challenging. For example, Docker Hub [4]—a popular public container registry—stores more than 2 million public repositories. These repositories have grown at the rate of about 1 million annually —and the rate is expected to increase which requires provisioning a large amount of new storage consistently. This puts intense pressure on the availability and scalability of Docker registry storage infrastructure because scaling-out involves data migration between existing storage serves and new storage servers, even among existing storage servers for load balancing, which largely hurts performance. In this paper, we propose a new Docker registry architecture, Sift, that integrates caching and deduplication with Docker registries to help reduce the storage requirements while mitigating any performance overhead.

Containers are running instances of *images* that encapsulate complete runtime environment for an application. An image comprises a set of shareable and content addressable *layers*. Each layer is made up of a set of files that are compressed in a single archive. Docker images/layers are stored in an online store called Docker registry [4] and accessed by clients. Since a layer is uniquely identified by a collision-resistant hash of its content, no duplicate layers are stored in the registry. Many container registries use remote cloud storage, e.g., S3 [2], as their backend storage system (Figure 1).

Is current registry deduplication effective? To guide our design, we downloaded and analyzed 47 TB (167 TB uncompressed) of Docker images from Docker Hub (containing over 5 billion files). We found that only 3% of the stored files across the layers are unique; the remaining are redundant copies. This clearly shows that the current layer-based sharing mechanism is unable to effectively remove data duplicates. This is despite the fact that many cloud storage systems implement deduplication to effectively eliminate redundant data. For example, Google cloud and AWS, employ in-line transparent data deduplication [11]. But the compressed layers reduce the opportunity for deduplication significantly. Effectively removing reduducncy from layers entails decompressing them before performing deduplication.

The challenge is that decompressing layers can have overheads besides the obvious decompression/compression. As shown in Figure 2, after decompression and simple file-level deduplication, the unique files may become scattered on multiple servers. To restore a layer, the layer's files would need to be first fetched from multiple servers, then compressed as a layer (to preserve transparency and avoid client-side API modifications) and sent back to the client. The extra network, I/O, and computation will slow down the response time for retrieving (pulling) an image.

Can caching help? Large container registry service providers such as Google and IBM use regional private registries across the world to facilitate a fast and highly-available service [1, 5]. This geographical distribution allows users to store images near their compute instances and experience a fast response time. An intuitive solution is to leverage these distributed registries as a cache to temporarily store popular layers and improve pull latency. The challenge here is that the layer access pattern is greatly different with traditional workloads since once a

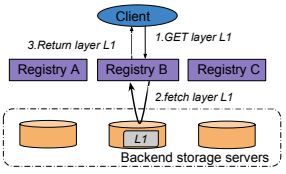


Figure 1: Without dedup.

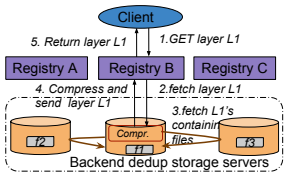


Figure 2: With dedup.

layer is pulled by a user, the layer won't be pulled again by this user because the layer is locally available to this user. This pattern has been observed by our registry workload analysis. The inter-access time for layer pulls can range from minutes to hours to months, making it difficult to manage the cache using traditional approaches such as LRU. Moreover, our Docker Hub image analysis shows that the layer sizes vary from a few MB to several GB and that a majority of layers sizes are around several MB. A small main memory cache cannot accommodate many layers, and thus is ineffective.

Sift to the rescue. Sift addresses the above challenges as follows. First, it explores layer access pattern and uses a user-access-history-based prefetch algorithm to accurately prefetch the layers that will be accessed shortly. Second, Sift embodies a two-tier heterogeneous cache architecture comprising a *layer buffer* and a *file cache* to hold prefetched layers. The layer buffer stores layers in main memory. The file cache stores the *deduped* unique files for the layers evicted from the layer buffer on a flash-based storage system. The mix of main memory and flash memory can realize the needed large capacity for caching prefetched layers and files. Our prefetch algorithm yields a hit ratio of 96%. Moreover, our two-tier heterogeneous cache is able to store 56% more layers into the file cache.

Sift use cases. on-prem registry. public-cloud registry.

2 Dataset analysis

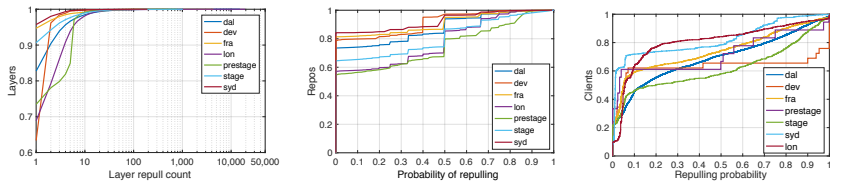
Before describing Sift, we must understand the trends in the access patterns of the Docker registries at the layer and repository level. INSERT DESCRIPTION OF DATASET HERE.

2.1 User access patterns

Figure 3(a) shows the CDF of layer repull count. Here, *repulling* indicates the act of pulling layers that have been pulled by the user before because they are no longer present on the user side. We see that majority of users don't *repull* layers frequently. For `Syd`, only 4% of layers are repulled by the same clients. `Dev` has the highest repull layer ratio of 36% while 83% of the repull layers are only repulled twice. Majority of repulled layers are repulled infrequently. For example, only 3% of layers from `Syd` are repulled more than twice. Layer from `Prestage` and `Lon` have the highest repull frequency. 5% of layers are pulled more than 6 times. We also observe that few clients *repull* layers continuously. The highest layer repull count is 19,300 from `Lon`. We think these clients probably deploy containers on a shared platform such as Cloud, and run ephemeral jobs such as stateless microservices. Once the applications are finished, the container images are automatically deleted. So when users launch containers again, they will repull layers again.

When different clients `pull` the same repository, they will fetch different amount of layers from the repository based on the contents of their local layer dataset. Even when the same clients `pull` the same repository at different times, they will fetch different amount of layers from the repository because their local layer dataset changes over time. Therefore, a `pull manifest` requests doesn't usually result in repulling the layers in the repository. Here, we define *repulling a repository* as repulling the layers in the repository for the same client. Figure 3(b) shows the CDF of the probability of repository repulling. The probability of repository repulling is calculated as the number of `pull manifest` resulting in repository repulling divided by the total number of `pull manifest` requests issued by the same client for the same repository. We see that majority of repositories aren't repulled. The repull repository ratio ranges from 15% for `Prestage` to 43% for `Prestage`. Majority of repull repositories have a low repulling probability. Only 20% of repositories from `Prestage`, `Stage`, and `Syd` have a repulling probability higher than 0.5. And only 20% of repositories from the rest 4 workloads have a repulling probability higher than 0.33. We also observe that few repositories' repulling probability are 1, meaning every time clients pull these repositories, they always repull the layers in these repositories.

Figure 3(c) shows the client repulling probability. Client repulling probability is calculated as the number of *repull* layer requests divided by the number of all `pull` layer requests issued by the same client. We see that majority of clients do repull layers but the probability is low. 60% of clients from `Prestage`, `Dev`, `Lon`, and `Fra` have a repulling probability lower than 0.1. 55% of clients from both `Dal` and `Stage` have a repulling probability lower than 0.1. Less clients have repulling probability range between 0.1 to 0.7. 10%-30% of clients have a repulling probability ranged from 0.2-0.7 across 7 workloads. We find few clients repull layers continuously. 2%-12% of clients have a repulling probability higher than 0.9 from workloads: `Dal`, `Dev`, `Fra`, `Prestage`, `Stage`,



(a) Layer repull count (b) Repository repulling probability (c) Client repulling probability

Figure 3: PDF of client repull count, repository repulling probability, and client repulling probability

Syd, and Lon.

2.2 Skewness and temporal access patterns

Figure 4(a) shows the CDF of layer popularity. We observe a heavy layer access skewness for Fra, Syd, Dal, Stage, and Lon. We see that 80%, 70%, and 60% of the pull layer requests access only 10% of layers, for Fra, Syd, Dal, Stage, and Lon respectively. Figure 4(b) shows the CDF of repository popularity. Compare to layer popularity, repository access skewness is heavier across 7 workloads. Almost 90% of pull layer requests access only 10% of repositories for Dev, Fra, Prestage, Syd, and Stage respectively. Almost 75% of pull layer requests access only 10% of repositories for both Dal and Lon. Figure 4(c) shows the CDF of client popularity. Dal, Dev, Fra, Lon, Prestage, and Stage shows a heavy client access skewness. 10% of clients send 95% pull layer requests for Lon. Syd shows a slight client skewness. 70% of requests are sent by 36% of clients. Overall, caching a layer with higher pull count will improve the hit ratio, especially for popular repositories and targeting active clients with higher repulling probability.

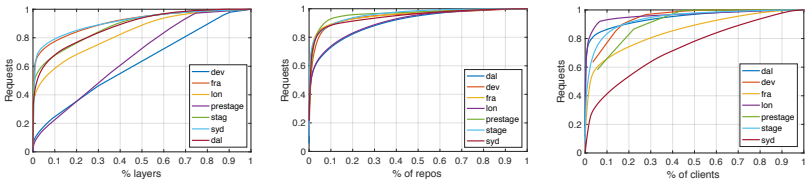
Next, we analyze the layer and repository reuse time. Layer reuse time means the duration between two consecutive requests to the same layer while repository reuse time means the duration between two consecutive *pull* manifest requests to the same repository. Figure 5(a) shows the CDF of layer reuse time. We see that layer reuse time distribution varies among different workloads. For Fra, Syd, and Stage, half of the layers' reuse time is shorter than 6 minutes. While half of layers from Dal and Lon have a reuse time higher than 1 hour. Half of layers from both Prestage and Dev are not accessed for over 100 hours. Consequently, for Dal, Lon, Prestage, and Dev, it may take longer than 1 hour for at least half newly requested layer to get a hit. These layers or the slices for them are unnecessarily for caching since their reuse time is too long and may cause other useful layers or slices to be evicted, called *cache pollution*. Figure 5(b) shows the CDF of repository reuse time. We see that repository reuse time is much shorter than layer reuse time. 80% of repositories are requested within 2-12 minutes across the 7 workloads. Figure 5(c) shows the CDF of client access intervals. client access interval means the duration between two consective requests issued by the same client. Client access intervals are much shorter than repository reuse time. 80% of client are active within 1 - 3 minutes for the 7 workloads. Hence, to eliminate *cache pollution*, we consider the reuse time of layer and repository as well as client access intervals during cache eviction.

3 Sift Design

In this section, we present an overview of Sift. Then, we present how Sift performs layer deduplication and layer restoring. Finally, we present our user-behavior based layer preconstruct diskcache.

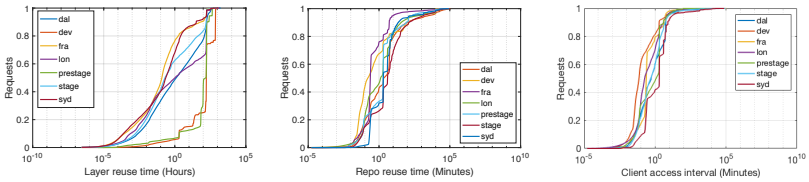
3.1 Overview

Sift provides different *deduplication modes* by deduplicating different number of layer replicas (see 3.2). Figure 6(a) shows a simple deduplication mode of Sift registry cluster compared with original registry cluster. As shown, upon a push layer request, registry *A* in original registry cluster stores the layer locally, denoted as *primary layer replica*. Meanwhile *A* replicates the layer to other two servers: *B* and *C*, denoted as *backup layer replicas*. The pull layer request can be served by any of the three layer replicas. Similar to original registry cluster, Sift first stores three layer replicas for newly pushed layers on three different servers: *A*, *B*, and *C*. Then, one layer replica is *deduplicated* into unique files, denoted as *primary file replicas*. After that, each unique file is replicated and stored on two different servers: *B* and *C* (denoted as backup file replica). In the end, two layer replicas are discard from server: *B* and *C*. pull layer request is sent to registry *A* which stores the primary layer replica. When registry *A* crashes, Registry *B* or *C* can restore a layer for incoming pull layer request. In this case, Sift has the same level of redundancy.



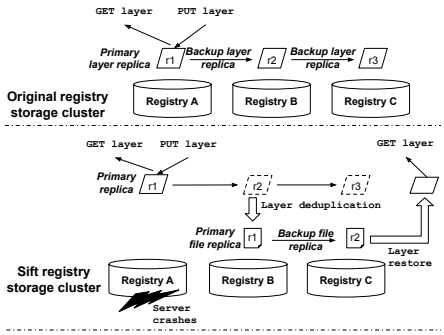
(a) Layer repull count (b) Repository repulling probability (c) Client repulling probability

Figure 4: CDF of probability for layers, repositories, and clients.

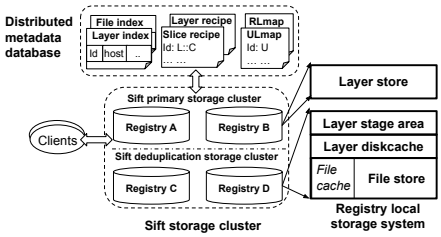


(a) Layer repull count (b) Repository repulling probability (c) Client repulling probability

Figure 5: CDF of reusetime for layers, repositories and clients' access intervals.



(a) Sift *VS.* original registry



(b) Sift storage system

Figure 6: Architecture of Sift.

Figure 6(b) shows the architecture of Sift. Sift comprises a cluster of registry servers and a distributed metadata database. Registry servers store and serve both manifest and layer requests sent by clients. Registry servers are divided into two groups: *primary cluster* and *deduplication cluster*, denoted as *P-servers* and *D-servers*. P-server mainly stores layer replicas and manifest replicas. While D-server deduplicates backup layer replicas and stores file replicas.

As shown in Figure 6(b), each D-server's local storage system is divided into three parts: layer stage area, layer diskcache, and file store. The *Layer stage area* temporarily stores newly added backup layer replicas as shown in Figure 6(a). After a layer replica is `deduplicated`, the remaining unique files are stored as primary file replicas in a content addressable *file store* with a small *file cache*. Meanwhile, each unique file is replicated to the peer servers to maintain the same level of redundancy. The layer replica is then deleted from layer stage area. To improve layer restoring performance, a layer replica is evenly divided into several **slices** and distributed across different registry servers so that the layer can be restored in parallel (see section 3.3).

Layer diskcache on a D-server is an on-disk layer cache which caches popular layers for later layer accesses to save restoring latency. When a P-server crashes, the `pull` layer request will be served from a D-server. If the `pull` layer request hits in D-server's layer stage area, the request will be served from layer stage area. Upon a `pull` layer request miss, D-server will rebuild the layer from file store based on its associated **layer recipe**(see section 3.4). Sift also preconstructs layers and caches them in the layer diskcache to improve the cache hit ratio (see section 3.5). The layer diskcache is a write-through cache; when layers are evicted from the diskcache, they are simply discarded.

Deduplication related metadata such as layer recipe, **slice recipe**, **layer index**, and **file index** is kept in a *distributed database* (Figure 6(b)) for reliability, consistency,

and fast accesses. Slice recipe and layer recipe are used to restore slices and rebuild layers. Layer index and file index records *content fingerprints* that are uniquely mapped to the associated physical layers and files respectively. Moreover, Sift keeps track of user accesses and repository status, denoted as **ULmap** and **RLmap**, and saves them into the distributed metadata database.

In the following, we first describe different deduplication modes provided by Sift. Then, we describe how Sift performs layer deduplication and layer restoring.

3.2 Deduplication modes

To satisfy different space saving and performance requirements, Sift provides two kinds of deduplication modes: Basic deduplication modes and selective deduplication modes, denoted as *B-modes* and *S-modes*. B-mode mantains a certain amount of layer replicas for each layer and deduplicates the rest layer replicas. While s-mode mantains a number of layer replicas for different layer proportionate to their popularity.

3.2.1 Basic deduplication modes

Assume that registry storage system uses R-way replication. Consider that layer dataset deduplication analysis [**?**] shows a file-level deduplication ratio of 2×. Thus, the average layer deduplication ratio for the layer dataset [**?**] is 2×.

B-mode r is defined as keeping r layer replicas and deduplicating the rest of layer replicas, where $r \leq R$. The space saving is calculated by following equation 1.

$$Spacesavings = \frac{R - (r + \frac{1}{2}(R - r))}{R} = \frac{R - r}{2R} \quad (1)$$

Layer pulling performance largely depends on registry server load. Assume that the total number of P-servers and D-servers are *P* and *D* especially. The total `pull` layer request load is *L*. Consequently, the average server load is $\frac{L}{P}$ and $\frac{L}{P+D}$ for Sift and original registry respectively. Note that original registry is Sift's B-mode *R*, which represents the best pulling layer performance with no space savings. The performance degradation can be calculated as:

$$Performancedegradation = \Delta(\frac{L}{P} - \frac{L}{(P+D)}) \times \delta_{R-r} \quad (2)$$

where $\Delta(\frac{L}{P} - \frac{L}{(P+D)})$ denotes the performance degradation caused by server load increases. δ_{R-r} means the performance degradation caused by load variance among different servers impacted by replication level.

3.2.2 Selective deduplication modes

In S-mode, the number of kept layer replicas r is proportional to their popularity ($r \leq R$). Layer popularity is calculated as $\frac{l_{pcnt}}{L}$, where l_{pcnt} is the total number of pulling request count to layer l. Consequently, the number of kept layer replicas r_l for layer l is $r_l = R \times \Phi(\frac{l_{pcnt}}{L})$, where $\Phi(\frac{l_{pcnt}}{L})$ denotes a replication factor.

The space savings for S-mode is calculated as following equation 3.

$$Spacesavings = \frac{R - (\sum_{l=1}^{n}(r_l + \frac{1}{2}(R - r_l)))}{R} \quad (3)$$

The performance degradation is:

$$Performancedegradation = \Delta(\frac{L}{P} - \frac{L}{(P+D)}) \times \prod_{l=1}^{n} \delta_{R-r_l} \quad (4)$$

where $\prod_{l=1}^{n} \delta_{R-r_l}$ means the performance degradation caused by load variance among different servers impacted by different replication levels.

3.3 Layer deduplication

As with the traditional Docker registry, Sift maintains a *layer index* to address the corresponding layers stored in the system. 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. The layer fingerprint is calculated by hashing the layer content, which is also used as layer identifier denoted as **layer id**. After that, Sift replicates r layer replicas on P-servers. Meanwhile, Sift submits $R - r$ layer replicas to D-servers. After D-servers receive the layer replicas, Sift will update *RLmap* with the layer and its associated repository, where RLmap maps a **repository id** (i.e., repository name) to its containing layers as shown in Figure 7.

Sift initiates a lightweight layer deduplication process which collaborates with the metadata database to discard redundant files from layers. The deduplication process has three major steps: layer decompression and unpacking, file-level deduplication, file replication, and **layer partitioning**. The first two steps are necessary for removing duplicate files from compressed layer tarballs. The last two step – file replication and layer partitioning is to evenly distribute the I/O and computation load of layer restoring across multiple registry servers while maintain the same redundant level with layer replication. Note that only one layer replica is selected among $R - r$ layer replicas to do layer deduplication. After that, all the layer replicas will be discarded.

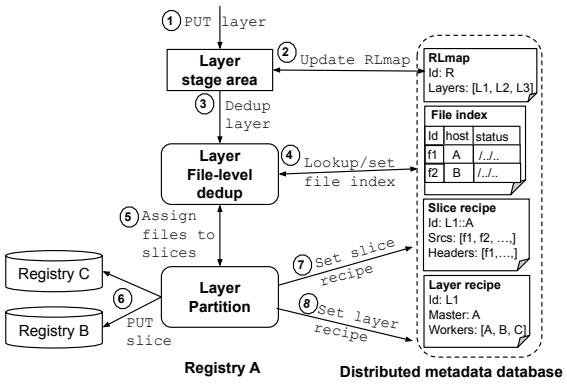


Figure 7: Layer deduplication and partitioning.

Layer file-level deduplication As shown in Figure 7, following the staging area, the layer is loaded to the layer deduplication process. In this process, the layer is first decompressed and unpacked into files. Then, the process computes a *fingerprint* for every file denoted as **file id**, and checks every file's fingerprint for presence in the file index. If the file is already stored, it will be discarded. Otherwise, it will be kept in file store while Sift records its *file id* to the file index.

File replication and layer partitioning After discarding duplicate files, the deduplication process distributes the layer's remaining files across different registry servers so that each server is able to rebuild an ∼equal-sized slice of the layer from its local file store. Then, the process sends slices to the corresponding registries. Finally, Sift will add slice recipes and layer recipe to metadata database as shown in Figure 7.

Figure 8 shows an example of file replication and layer partition. Each file has two replicas distributed on different servers, *r1, r2, r3,* and *r4* are four primary file replicas that were already stored in the system. and *r1', r2', r3',* and *r4'* are their associated backup file replicas. A layer *L* that contains files *r1, r2, r3, r4, r5, r6, r7,* and *r8* is removed. Its containing files *r1, r2, r3,* and *r4* are deduplicated while unique files *r5, r6, r7,* and *r8* will be added to the system, As shown, *r5, r6, r7,* and *r8* are replicated and distributed to registry *A, B, C,* and *D* as primary file replicas and backup file replicas. Consequently, primary file replicas *r1* and *r5* stored on registry *A* comprise one of layer *L*'s partitions, denoted as **primary slice**: *layerid:A*. The rest two backup file replicas *r4'* and *r8'* stored on registry *A* comprise a **backup slice** for primary slice stored on registry *D* (i.e., *r4* and *r8*).

Algorithm 1 details layer deduplication and partitioning algorithm. After layer decompression, each file entry in the layer archive is represented as a **file header** and a **file content**. The file header contains the file name, path, size,

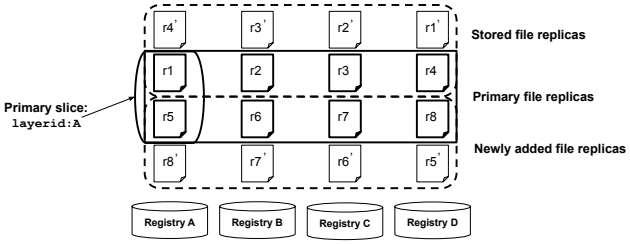


Figure 8: Layer replication and partitioning.

Algorithm 1: Layer deduplication and partitioning

Input:
$FileIndex$: File index.
Output:
$LayerRecipe$: Layer recipe for layer L.
$SliceRecipes$: Slice recipes for layer L' slices.

```
1  archive ← Decompress layer L
2  foreach header, content in archive do
3      Id ← Hash content
4      if Id in FileIndex then
5          host ← FileIndex[Id].host
6          src ← FileIndex[Id].header
7          /* SliceRecipe is identified by 'layerid::host'
           SliceRecipe[L :: host] ← Add (header, src)
8          /* Skip redundant file.
9      else
10         /* create file with content in file store.
11         file ← Create and write content
12         src ← Stat file
13         entries ← (header, src)
14 do
15     /* get file entry with maximum file size.
16     maxFile ← Max entries
17     /* get smallest slice.
18     minSlice ← Min SliceRecipes
19     /* assign biggest file to smallest slice.
20     minSlice ← Add maxFile
21     /* Set maxFile to FileIndex
22     /* Remove maxFile from entries.
23 while entries;
24 /* Send slices.
25 /* Set SliceRecipes.
26 /* Include all slices' hosts to workers.
27 /* Select biggest slice' host as master.
28 /* Set LayerRecipe.
```

mode, owner information, etc. The file header is needed to rebuild the *file header* for the associated file entry in the layer archive.

Sift records each file entry's file header and calculates the file id by hashing the file content. As mentioned earlier, the file index maps a *file id* to its associated physical file that is stored in the file store. To address a physical file, each file id in the file index holds its *host address* and **file status** of the physical file which mainly contains file name, path, and size. As shown in Algorithm 1, if a file's file id already exists in file index, this file will be added to its host's corresponding partition (i.e, slice). Otherwise, Sift stores the file content as a physical file in the file store and retrieves the file status of the physical file. The file index is updated accordingly.

The slice recipe is identified by a simple combination

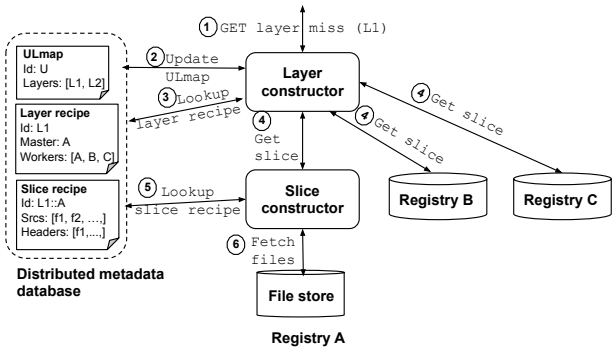


Figure 9: Parallel layer restoring.

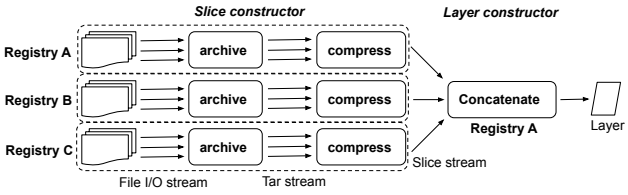


Figure 10: Parallel streaming layer construction.

of the layer id and its host registry address, denoted as $layerid :: host$ as shown in Algorithm 1. A slice recipe represents a layer partition and is used to construct a partial layer, which records each file entry' $header$ in the layer archive partition and its corresponding physical file's status src. In this case, each file is represent as a $header$ in the layer archive and its corresponding physical file's status src.

To distribute the newly added unique files' primary replicas, Sift uses a greedy partition algorithm to assign the biggest file to the smallest primary slices such that layer can be evenly partitioned among registry servers. All the primary slices' hosts are denoted as layer restoring **primary workers**. Next, Sift stores slice recipes in metadata database after successfully sending slices to their corresponding workers. Sift selects the biggest slice' host as layer restoring **primary master**. Layer recipe records the primary workers and master information for layer restoring. After that, Sift distributes primary slices' corresponding backup slices. Their hosts are denoted as **backup workers**. Finally, Sift stores layer recipe in metadata database.

3.4 Layer restoring

When a `pull` layer request is received and P-servers fail, Sift will first search layer diskcache on D-servers. If not found, Sift will rebuild the layer from file store according to its layer recipe.

Parallel layer construction When a `pull` layer request is received, Sift will update *ULmap*. ULmap records user access status, which maps a **user id** to its accessed layers with its corresponding access count, where user id is defined as client request address.

Upon a `pull` layer request miss, Sift initiates a layer restoring process for it. The process has two parts: slice constructor and layer constructor. Figure 9 shows an example of parallel layer restoring when a `pull` layer request miss happens. First, layer constructor fetches the layer recipe from metadata database. As shown in $L1$'s layer recipe, the restoring workers contains registry A, B, and C. Since registry A is the restoring master, it sends "`Get slice`" requests to its peer workers: B and C. After a `Get slice` request is received, B and C start slice construction and return a slice back to A respectively. Meanwhile A instructs local slice constructor to rebuild a slice for $L1$. Slice constructor first gets its associate slice recipe from metadata database keyed by a combination of layer id and registry address ("$L1 :: A$"). Then, based on slice recipe, slice constructor fetches files pointed by $Srcs$ and builds a slice archive. After receiving all slices, layer constructor concatenates them into a compressed layer and sends back the client.

Streaming layer construction Figure 10 details the stateless streaming layer construction process. First, slice constructor loads file in parallel from file store based on the slice recipe. Each file is written to an archive buffer asynchronously. Before writing file content to the archive buffer, slice constructor first builds and writes its associated file header into the archive buffer according to the slice recipe. After archiving all the files in the slice, the archive buffer will be divide into few chunks and compressed in parallel, then concatenated into a single compressed slice stream. Through network transfer, multiple slice streams will be concatenated into a single layer. No intermediate file will be created or stored on disk.

Sift uses a small file cache to reduce file I/Os. File cache uses LFRU replacement policy.

3.5 Layer preconstruction

Based on the observations made in xxxx, Sift preconstructs layers before layer accesses to save layer construction time.

User behavior based layer preconstruction Figure 11 shows how to preconstruct layers for later accesses. When a `GET manifest` request is received by registry A, it gets the requested repository information and client information from RLmap and ULmap respectively, and computes a list of target layers that will be accessed later by

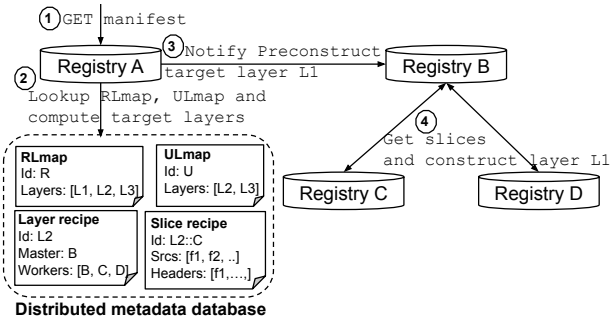


Figure 11: Layer preconstruction.

Algorithm 2: Layer preconstruction

Input:
θ_{rpc}: Threshold for repull layers to be preconstructed.
$RLmap$: Repository to layer map.
$ULmap$: User to layer map.
$LayerRecipes$: Layer recipes.
$SliceRecipes$: Slice recipes.

```
r ← request received
if r = GET manifest then
    difference ← RLmap[r.repo] − ULmap[r.client]
    intersection ← RLmap[r.repo] ⋂ ULmap[r.client]
    targets ← difference
    foreach layer in intersection do
        if layer.rpcnt > θ_rpc then
            targets ← layer
    foreach layer in targets do
        master ← LayerRecipes[layer].master
        Notify master 'PRECONSTRUCT layer'
else if r = PRECONSTRUCT layer then
    if r.layer not in DiskCache then
        workers ← LayerRecipes[r.layer].workers
        foreach worker in workers do
            slices ← GET slice from worker 'GET slice'
        layer ← Concatenate slices
        DiskCache ← layer
        /*Set timer for layer/*
else if r = GET slice then
    slice ← Construct by following SliceRecipes[r.slice]
    Serve slice
```

this client. As shown, to preconstruct target layer $L1$, registry A gets $L1$'s layer recipe from metadata database and sends a notification of `preconstruct layer` $L1$ to $L1$'s restoring master: registry B. After receiving the notification, B sends `get slice` requests to its peer workers: C and D. When B, C, and D finished slice constructions for L, three slices are concatenated by B and saved in B's layer diskcache.

Algorithm 2 shows how to determine target layers based on observed user access pattern. When a `GET manifest` request *r* is received, Sift gets a set of layers associated with the requested repository *r.repo* from RLmap, denoted as $RLmap[r.repo]$. Meanwhile, it also gets a set of layers associated with the client *r.client* from ULmap, denoted as $ULmap[r.client]$. After that, it calculates a list of layers in the requested repository that have not been `pulled` by the client by computing the difference set between $RLmap[r.repo]$ and $ULmap[r.client]$ (denoted as $difference$). All the layers in $difference$ are included in target set. Sift also calculates a intersection set between $RLmap[r.repo]$ and $ULmap[r.client]$ (denoted as $intersection$). The layers in $intersection$ are already pulled by client. To determine whether a layer in $intersection$ will be $repulled$ by client, Sift compares its repull count against a predefined threshold θ_{rpc}. If the layer's repull count is greater than θ_{rpc}, which means this layer will be repulled with high probability. Thus, it is included in target layer set.

For each layer in target set, Sift gets the restoring master from its layer recipe, and sends a notification of `preconsturct layer` to the master. After the master receives the notification, it first check if the requested layer is in the diskcache. If not, it starts layer construction and saves the preconstructed layer in diskcache.

Layer diskcache eviction To exploit the temporal trend of clients and repositories, Sift sets timer for each cached layers. Once a layer is expired, it will be simply deleted from diskcache.

Layer diskcache replacement is triggered when the free space in diskcache is lower than threshold θ_C. As shown in Algorithm 3, Sift maintains a LFRU list [**?**] of cached layers to exploit layer temporal trend. If the free space is low, Sift selects the least frequent recently used layer to evict.

Algorithm 3: Eviction

Input:
θ_C: Capacity threshold for diskcache to trigger replacement.
$LFRU$: LFRU list of layers.
θ_t: Expiration time determines for how long the layer stays in diskcache.

```
foreach layer in LFRUlayer do
    /*Remove expired layer */
    if layer.elapsed > θ_t then
        Evict layer
while freeSpace < θ_C do
    /*Replacement */
    layer ← LFRUlayer.last_item()
    Evict layer
    freeSpace + = sizeof layer
```

4 Implementation

Metadata database implementation We use Redis [**?**] as metadata database to save slice and layer recipes, file index, layer index, ULmap, and RLmap. To guarantee the durability of metadata, we configure Redis to use an *append-only file (AOF)* to log all changes. Thus, when Redis server stops, we can restart Redis and re-play the AOF to rebuild the dataset to prevent data loss. Moreover, we configure Redis to save RDB (Redis Database File) snapshots every few minutes to further improve metadata reliability. To improve metadata availability, we configure

Redis to use three replicas. In this case, Redis will not become a performance bottleneck.

Distributed lock for distributed deduplication We use Redis as a distributed lock to make sure that no file duplicate is stored in registry cluster. For file index, layer index, slice and layer recipes, each key can be set to hold its value only if the key does not exist in Redis database (i.e, SETNX [**?**]). When key already holds a value, a file duplicate or layer duplicate is identified and will be removed from registry cluster. Besides, each registry instance maintains a synchronization map for layer restoring to make sure that only a unique layer is restoring at a time. While different layers can be constructed in parallel. In this case, redundant layer restoring can be avoid to save I/O and network bandwidth, and computation resources. Slice restoring is the same.

GET-DEDUP LOCK for fast layer dedup. layer stage area delay 1-2 seconds. **GET-RESTORING LOCK** for fast layer restoring. sync.map and delay 5 seconds, and buffer Stage area to layer cache

Layer deduplication implementation During layer deduplication, only *regular* file duplicates are removed. Sift does not deduplicate *irregular* files. After removing regular file duplicates, the remaining unique files are saved in a local file store. File store is implemented by modifying Docker registry's file system driver [**?**], which is local storage driver that uses a directory tree in the local file system. The default root directory in the local file system for registry storage system is '/var/lib/registry' (denoted as $rootdir$). Each layer is stored in directory '$rootdir$/docker/registry/v2/blobs/sha256/⟨first two hex bytes of layer digest⟩/⟨hex layer digest⟩/' (denoted as $layerdir$) as a file named 'data'. We store the unique files for the layer in the same directory with layer as '$layerdir$/uniquefiles/⟨hex file digest⟩'.

Layer restoring implementation To speedup layer restoring performance, we utilize a parallel gzip compression/decompression library (pgzip) [**?**] to do layer or slice compression and decompression. Different from single-threaded standard gzip compression library [**?**], pgzip splits compression stream into blocks so that data can be compressed in parallel to speedup compression. pgzip is compatible with standard gzip library.

File cache and layer diskcache implementation File cache is implemented on bigcache [**?**], which can provide fast and concurrent cache access. Layer diskcache is implemented by using diskv [**?**], which is a disk-backed key-value store. However, both bigcache and diskv do not use any cache algorithm. Note that bigcache always evicts the oldest entries from cache without consider the entries' popularity. We build two ARC lists for bigcache and diskv respectively by using a golang cache library [**?**], which provides expirable cache with different cache algorithm, such as LFU, LRU and ARC [**?**]. Note that we select ARC with expiration as our cache algorithm. We combine ARC cache algorithm with bigcache as our file cache and ARC cache algorithm with diskv as our layer diskcache.

Client implementation Client is comprised of a trace replayer and a proxy emulator. The trace replayer is implemented by modifying IBM cloud registry trace replayer [**?**]. Instead of generating a random layer [**?**], our trace replayer first matches IBM cloud registry trace with the real layers from our dataset downloaded from Docker Hub [**?**, **?**]. Specially, we extract layer digest from layer request recorded in IBM traces [**?**]. Then, we randomly match each layer digest to a layer in our dataset. Consequently, each layer request from IBM traces `pulls` or `pushs` a real layer. For manifest requests recorded in IBM traces, we generate a random file to emulate a manifest file. This is because sift do not deduplicate manifest file. The trace replayer uses Docker registry client emulator Python-dxf [**?**] to `pull` or `push` layers or manifests to registry.

In addition, client uses a proxy emulator to decide the destination registry server for each request. The proxy emulator uses consistent hashing algorithm (CHT) [**?**] to distribute layers and manifests. Proxy emulator maintains a CHT 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, proxy emulator also uses CHT to calculate the destination server. While for `pull` layer requests, proxy emulator will first consult the metadata database to get the layer recipe. If the recipe presents, the emulator will send the request to the restoring master. Otherwise, the emulator will calculate the destination server by using CHT, and send the request to the destination server.

Redundancy implementation To maintain reliability and fault-tolerance, traditional registry maintains a certain level of redundancy for each layer by delegating replication to backend distributed storage systems, such as S3 and Swift [**?**]. To implement the same level of redundancy for layers, Sift storage system stores R replicas for each layer on R registry servers by using chain replication [**?**]. In this case, each layer has a primary replica and $R-1$ backup replicas. During layer deduplication, only primary replica is dedupcated. After that, the remaining unique files will be R-way replicated. Then the layer will be partitioned to $N*R$ servers denoted as $N*R$ slices instead of N slices(see Section [**?**]). Here, each slice for the layer

has a primary replica and $R-1$ replicas. Consequently, each unique file stored in cluster has R replicas. For layer restoring, each layer has a primary restoring master and $R-1$ backup restoring master. After layer deduplication, the primary layer and its two backups are discard.

5 Evaluation

We first present our testbed and workloads. Then we present our evaluation results.

5.1 Testbed

Our testbed includes three clusters: 5-node client cluster (Amaranths), 8-node registry cluster (Hulks), and 24-node registry cluster(Thors). Each client Amaranth server is equipped with 8 cores, 64 GB RAM, 1 TB HDD, and 1000Mb/s NIC. Each registry Hulk server is equipped with 32 cores, 64 GB RAM, 500 GB SSD, 1 TB HDD, and 10,000Mb/s NIC. Each registry Thor server is equipped with 8 cores,16 GB RAM, 500 GB SSD, and 10,000Mb/s NIC.

5.2 Workloads and dataset

[**?**] presents an comprehensive analysis of our whole dataset downloaded from Docker Hub. Docker image popularity distribution shows a heavy skewness [**?**]. To evaluate how our sift works for frequently accessed images, we select a subset of 74,000 popular images (i.e., image with a pull count greater than 100) from the whole dataset as our evaluation dataset, totally 12.5 TB with 507,023 layers. After decompressing, the total size of our dataset is 27.7 TB. With file-level deduplication ratio applying to our decompressed dataset, the total size of unique files is only 7.2 TB, with a dedup ratio of 0.42.

[**?**] presents IBM cloud registry workload traces spanning ∼80 days for 7 different registry clusters. To evaluate how our sift performs for real-wold workloads, we emulated a real-world workload replayer by randomly matching IBM traces [**?**] and our dataset(see section [**?**]).

5.3 Performance

5.3.1 Overall performance

5.3.2 Breakdown performance

5.3.3 Performance Comparison

5.3.4 Performance with different parameters

Layer size impact

Cache parameter impact

Registry cluster size impact

Repull threshold

Compression level

5.4 Deduplication performance

6 Related Work

A number of studies investigated various dimensions of Docker storage performance [16, 17, 19, 27, 31, 34, 32, 19]. Anwar et al. [14] performed a detailed analysis of an IBM Docker registry workload but not the dataset. Data deduplication is a well explored and widely applied technique [21, 25, 28, 30, 36]. A number of studies which focus on real-world datasets [20, 22, 23, 26, 29, 33, 35] can be complementary to our approach.

7 Conclusion

We presented Sift, a new Docker registry architecture that integrates caching and deduplication to improve the registry's performance and decrease the storage capacity requirement. Sift leverages a two-tier heterogeneous cache architecture to efficiently improve cache space utilization. Our prefetch algorithm can improve the cache hit ratio and mitigate the overhead of decompression-enabled deduplication.

Table 1: Workload parameters

Trace	♯ Reqs	♯ GET L	♯ GET M	♯ PUT L	♯ PUT M	L dataset (GB)	♯ Uniq L	♯ Uniq M	Duration (Hr)
Dal	5000	2867	2000	124	9	6.6	1278	88	6
Dev	5000	980	3987	29	4	4.7	872	93	60
Fra	5000	1602	3278	111	9	2.4	420	43	24
Lon	5000	924	3972	98	6	4	698	88	6
Pre	5000	3901	1016	79	4	20.2	3908	284	11
Sta	5000	2585	2306	99	10	2.8	502	64	6
Syd	5000	1310	3653	35	2	0.9	154	18	30

8 Discussion

While the performance aspects of Docker containers on client side have been studied extensively, we believe the current registry design is inefficient and will become a bottleneck when handling the expected massive amount of images. In this work, we have attempted to solve the registry redundant data issue using an integrated caching and deduplication approach that is guided by user access patterns. Our approach showcases the benefits for both the Docker registry service providers as well as the clients.

We believe that our paper will lead to discussion on the following points of interest: (1) The need for a deduplication approach tailored for Docker registry. Since deduplication is a well-studied area, we believe that the paper will lead to hot discussion on how the extant approaches can be applied/adapted to diverse data formats such as compressed files. (2) Issues of how to effectively manage a layer/file cache by exploiting user access patterns, and where to place the cache (e.g., at registry side, remote cloud storage side, or Docker client side) to benefit overall container performance. (3) The role of layer and image. The redundant data issue stems from containers' storage virtualization technique that use union file systems to pack everything inside different layers (that are used as a plugin OS image). An open aspect of our work is improving the efficiency of current union file systems and container storage virtualization technique.

An unexplored issue is redundant data on the Docker client side, which differs greatly from registry storage. This is because the registry stores compressed layers, while client-end host-storage system stores uncompressed layers. Therefore, the client-side redundancy issue under limited local storage space is just as bad or worse as the redundancy in the registry storage system. However, applying current deduplication methods—that are usually deployed on backup storage systems—on the client side would greatly impact the container runtime performance. This is because performing file-level deduplication on the Docker client side requires intensive file fingerprint calculations, and file fingerprint lookup, which will slowdown performance. Furthermore, we believe that our work can help synchronize deduplication for both client side deduplication and registry side deduplication to yield desirable win-win solutions.

References

[1] About ibm cloud container registry. https://cloud.ibm.com/docs/services/Registry?topic=registry-registry_overview#registry_overview.

[2] Amazon s3. https://aws.amazon.com/s3/.

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

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

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

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

[7] Openstack swift. https://docs.openstack.org/swift.

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

[9] Registry as a pull through cache. https://docs.docker.com/registry/recipes/mirror/.

[10] Squid Cache - Optimising Web Delivery. http://www.squid-cache.org/.

[11] Storreduce. http://storreduce.com.

[12] The Plasma In-Memory Object Store. https://arrow.apache.org/docs/python/plasma.html.

[13] Varnish HTTP Cache. https://varnish-cache.org/.

[14] 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*.

[15] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBLER, T., WEI, M., AND DAVIS, J. D. CORFU: A shared log design for flash clusters. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)* (San Jose, CA, 2012), USENIX, pp. 1–14.

[16] 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*.

[17] CANON, R. S., AND JACOBSEN, D. Shifter: Containers for HPC. In *Cray User Group'16*.

[18] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The google file system.

[19] 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*.

[20] JIN, K., AND MILLER, E. The effectiveness of deduplication on virtual machine disk images. In *ACM SYSTOR'09*.

[21] 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*.

[22] LU, M., CHAMBLISS, D., GLIDER, J., AND CONSTANTINESCU, C. Insights for data reduction in primary storage: A practical analysis. In *ACM SYSTOR'12*.

[23] 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*.

[24] MEISTER, D., KAISER, J., BRINKMANN, A., CORTES, T., KUHN, M., AND KUNKEL, J. A study on data deduplication in hpc storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), IEEE Computer Society Press, p. 7.

[25] MUTHITACHAROEN, A., CHEN, B., AND MAZIERES, D. A low-bandwidth network file system. In *ACM SOSP'01*.

[26] SHIM, H., SHILANE, P., AND HSU, W. Characterization of incremental data changes for efficient data protection. In *USENIX ATC'13*.

[27] SPILLANE, R. P., WANG, W., LU, L., AUSTRUY, M., RIVERA, R., AND KARAMANOLIS, C. Exoclones: Better Container Runtime Image Management Across the Clouds. In *USENIX HotStorage'16*.

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

[29] 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*.

[30] 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*.

[31] 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*.

[32] THALHEIM, J., BHATOTIA, P., FONSECA, P., AND KASIKCI, B. Cntr: Lightweight OS containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)* (Boston, MA, 2018), USENIX Association, pp. 199–212.

[33] 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*.

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

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

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