# Slimmer: Weight Loss Secrets for Docker Registries

Nannan Zhao[1], Vasily Tarasov[2], Ali Anwar[1], Lukas Rupprecht[2], Dimitrios Skourtis[2],
Amit S. Warke[2], Mohamed Mohamed[2], and Ali R. Butt[1]
[1]Virginia Tech, [2]IBM Research—Almaden

***Abstract*—Due to their tight isolation, low overhead, and efficient packaging of the execution environment, Docker containers have become a prominent solution for deploying modern applications. Docker registries store a large amount of images and with the increasing popularity of Docker, they continue to grow. For example, Docker Hub—a popular public registry—stores more than half a million public images. In this paper, we analyze over 167 TB of uncompressed Docker images and evaluate the potential of file-level deduplication in the registry. Our analysis reveals that only 3% of the files in images are unique and Docker's existing layer sharing mechanism is not sufficient to eliminate this profound redundancy. We then present the design of Slimmer—a Docker registry with file deduplication support—and conduct a simulation-based analysis of its performance implications.**

***Keywords*-Docker; Deduplication; Docker registry; Distributed storage systems;**

## I. Introduction

*Containers* have recently gained significant traction due to their low overhead, fast deployment, and the rise of container management frameworks such as Docker [1]. Polls suggest that 87% of enterprises are at various stages of adopting containers, and they are expected to constitute a $2.7 billion market by 2020 [2].

Docker combines process containerization with efficient and effective packaging of complete runtime environments in so called *images*. Images are composed of shareable and content addressable *layers*. A layer is a set of files, which are compressed in a single archive. Both images and layers are stored in a Docker *registry* and accessed by clients as needed. Since layers are uniquely identified by a collision-resistant hash of their content, no duplicate layers are stored in the registry.

Registries are growing rapidly. For example, Docker Hub, the most widely used registry, stores more than 500,000 public image repositories comprising over 2 million layers and it keeps growing. This massive image dataset presents challenges to the registry storage infrastructure and so far has remained largely unexplored.

In this paper, we perform the first large-scale redundancy analysis of the images and layers stored in Docker Hub. We downloaded 47 TB (167 TB uncompressed) worth of Docker Hub images, which in total contain over 5 billion files. Surprisingly, we found that only around 3% of the files are unique while others are redundant copies. This suggests that current layer sharing cannot efficiently remove data duplicates. Container images are similar with virtual machine images in the sense that they all serve as OS snapshots. Difference users might choose similar libraries and run similar applications, which incurs a considerable redundancy across different container images.

Given our findings, we propose Slimmer, a file-level content addressable storage model for the Docker registry. Slimmer unpacks layer tarballs into individual files and deduplicates them. When a Docker client requests a layer, Slimmer dynamically reconstructs the layer from its constituent files. To assess the feasibility of our design, we conduct a simulation-based evaluation of Slimmer.

## II. Deduplication analysis

In this section, we investigate the potential for data reduction in the Docker registry by estimating the efficacy of layer sharing, compression, and the proposed file-level deduplication.

### A. Methodology

To analyze the benefits of different data reduction techniques for the Docker registry, we downloaded a large number of Docker images from Docker Hub registry. Docker Hub does not provide an API to retrieve all repository names. Hence, we crawled the registry's website to obtain a list of all available repositories, and then downloaded the *latest* version of an image and its corresponding layers for each repository. We downloaded 355,319 images, resulting in 1,792,609 compressed layers and 5,278,465,130 files, with a total compressed dataset size of 47 TB.

### B. Data reduction analysis

*Layer sharing:* Docker supports the sharing of layers among different images. To analyze the effectiveness of this approach, we compute how many times each layer is referenced by images. Figure 1 shows that around 90% of layers are referenced by a single image, an additional 5% are referenced by 2 images, and less than 1% of the layers are shared by more than 25 images. From the above data we can estimate that without layer sharing, the Docker Hub dataset would grow from 47 TB to 85 TB, implying a **1.8**× deduplication ratio provided by layer sharing.

Figure 5. Off-line file-level deduplication run time.

## B. *Performance evaluation*

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

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

Figure 5 shows the CDF for each sub-operation of Slimmer. Unpacking, Decompression, Digest Calculation, and Searching are part of the deduplication process and together make up the Dedup time. Searching, Archiving, and Compression simulate the processing for a `pull` request and form the Pulling time.

*Push:* Slimmer does not directly impact the latency of `push` requests because deduplication is performed asynchronously. The appropriate performance metric for `push` is the time it takes to deduplicate a single layer. Looking at the breakdown of the deduplication time in Figure 5, we make several observations.

First, the searching time is the smallest among all operations with 90% of the searches completing in less than 4 ms and a median of 3.9 ms. Second, the calculation of digests spans a wide range from 5 $\mu$s to almost 125 s. This is because the time mainly depends on the layer size, i.e. the fewer and smaller files a layer contains, the faster it is to compute all digests for the layer. Typically, smaller layers contain a smaller number of smaller files, which takes much less time to calculate their digests. While if the layer is bigger, the digest calculation overhead will be higher. 90% of digest calculation times are less than 27 s while 50% are less than 0.05 s. The diversity in the timing is caused by a high variety of layer sizes both in terms of storage space and file counts. Third, the run time for decompression and unpacking follows an identical distribution for around 60% of the layers and is less than 150 ms. However, after that, the times diverge and decompression times increase faster compared to unpacking times. 90% of decompressions take less than 950 ms while 90% of packing time is less than 350ms.

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

From Figure 5 we can see that 55% of the layers have close compression and archiving times ranging from from 40 ms to 150 ms and both operations contribute equally to pulling latency. After that, the times diverge and compression times increase faster with an 90th percentile of 8 s. This is because compression times increase for larger layers and follow the distribution of layer sizes (see Figure 2). Compression time makes up the major portion of the pull latency and is a bottleneck. Overall, the average pull time is 2.3 s.

## IV. Conclusion

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

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

## References

[1] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, vol. 38, no. 5, 2005.

[2] 451 Research, "Application Containers Will Be a $2.7Bn Market by 2020," https://tinyurl.com/ya358jbn.