Implementing a Highly-Performant Dataset Sorting Framework on a Distributed Storage System to Accelerate AI Workloads

Andrew Liang

September 18, 2018

1 Introduction and Background

1.1 Introduction

During the summer of 2018, I worked at NVIDIA – a company credited for inventing the GPU and a world leader in artificial intelligence (AI) computing advancements [1]. As a software developer on the distributed storage engineering team, I worked on DFC: a distributed, persistent object cache written in Golang.

During my internship, I was challenged with the task of developing a mechanism for performing a sorting-and-resizing operation on a dataset of archive files stored within DFC. The goal of this feature was to offload this processing step commonly done on AI computing clients to the storage cluster, thus accelerating AI training workloads speeds.

1.2 DFC

DFC is a distributed object storage system and persistent cache tailored for AI applications [2]. A DFC cluster is composed of two types of nodes: proxies and targets. Proxies act as Hypertext Transfer Protocol (HTTP) gateways between DFC clients and targets. Furthermore, proxies are responsible for cluster-level management such as serializing cluster-wide metadata updates and leader election in case of proxy failover. DFC targets are responsible for managing the object storage lifecycle by operating on the targets' local disks and filesystems. Every DFC cluster has a *primary proxy*, which is the master node of the cluster.

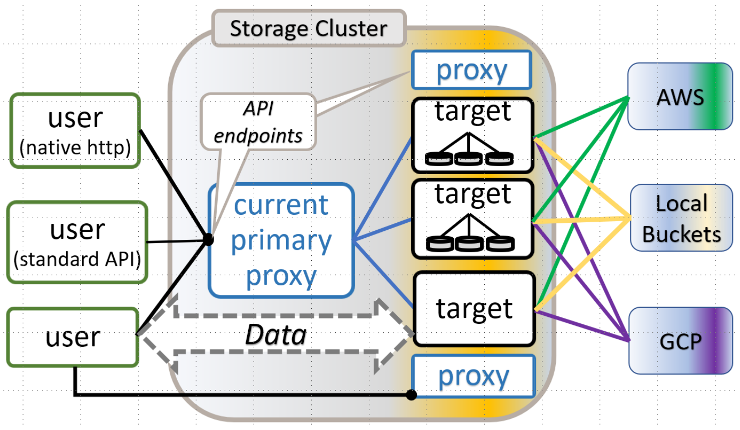


Figure 1: A high-level overview of DFC's architecture [2].

In DFC, an object's storage location in the cluster is decided by a consistent hashing algorithm called Highest Random Weight (HRW), which provides a uniform distribution of objects across all targets. A property of HRW is that if a target joins – or leaves – a cluster, only approximately

$1/N$ of all N objects in the cluster need to be rebalanced [3].

As a cache, DFC supports Google Cloud Platform (GCP) and Amazon Web Services (AWS) as cloud storage providers for cold reads and writes.

2 Problem Specification

2.1 Description

In AI training and inference workloads, there is a ubiquitous operation which involves shuffling or sorting a dataset in some deterministic order to optimize subsequent training of models [4]. For example, in Apache Hadoop's MapReduce framework, this corresponds to the shuffle phase. This sorting operation – which will be referred to as `SortDataset` – is recently becoming increasingly typical in petabyte-scale video and image datasets for deep learning [5]. These video and image datasets are often comprised of compressed or uncompressed archive files, for example tarballs, MessagePack files, and ZIP files. These archive files, or shards, contain contents of other files along with some associated metadata. Performing `SortDataset` on a dataset D_1 comprised of archive files shard-00000.tgz, shard-00001.tgz, . . . , shard-99999.tgz would result in a new dataset D_2 which is composed of the file contents, or records, of D_1 in some defined sorted order. Another aspect of `SortDataset` allows for the size of each new shard to be different from the size of each original shard in the initial dataset. For instance, consider a dataset D 100TB in size composed of 100,000 shards of size 1GB each. Performing `SortDataset` on D with an expected shard size of 100MB would result in a new dataset composed of 1,000,000 shards. By reducing the size of each shard, more parallelism can be achieved on subsequent processing, leading to greater computing efficiency.

Consider a situation as depicted in Figure 2 below, in which an AI computing client is processing a training workload. It interfaces with a petabyte-scale storage cluster through a 100 Gigabit Ethernet (100GbE) network switch.

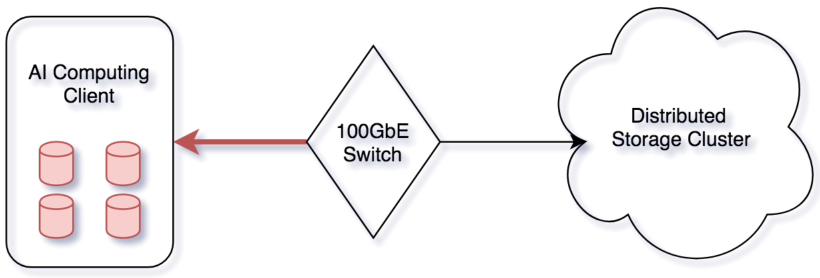


Figure 2: Pressured components (shown in red) result in performance bottlenecks at scale.

If the AI client is responsible for `SortDataset`, two major performance bottlenecks will be encountered during processing: the Network Interface Controller (NIC) during the data ingestion phase, and the system resources of the client. As a result of all the file manipulations required to perform `SortDataset`, memory and disk usage of the client can easily reach a high watermark and become a performance bottleneck when operating at scale.

Now, consider an alternative where the system responsible for performing `SortDataset` is not the AI client, but rather the storage cluster. By moving this processing closer to the source of the data, pressure from ingress network traffic no longer becomes a concern, and major performance optimizations are achieved by taking advantage of data locality. Furthermore, cluster-wide system resource bottlenecks can be resolved well by scaling out the cluster horizontally using low-cost commodity hardware. Finally, since the storage cluster is distributed, the processing required for `SortDataset` can be maximally parallelized to achieve optimal performance.

2.2 Existing Solutions

Currently, Google's Google File System (GFS) and Amazon's Simple Storage Service (S3) represent the state of the art in large-scale data processing [5]. However, there are two main things to consider with regards to using these technologies to solve the specified problem. Firstly, GFS and S3 are proprietary technologies and closed-source, meaning users cannot freely modify the source code of these services. This restriction ultimately sets a limit to how much product customization is possible. Furthermore, because these technologies are proprietary, they are unavailable for independent commercial deployment. This fact means that a different company such as NVIDIA would not be able to deploy their own clusters running GFS or S3. This limitation may be problematic if extreme data privacy or security is a concern.

In the open-source world, there exist tools like Hadoop and Spark for these types of processing requirements. However, these tools were designed in a previous era – prior to the existence of technologies like containerization, and before petabyte-scale deep learning [5]. As a result, these tools are also not fully satisfactory as solutions.

Finally, none of these technologies were architected specifically for use by AI applications. For feature flexibility, to take advantage of DFC's design tailored for AI workloads, and for DFC to continue evolving as a solution targeting AI applications, an implementation for `SortDataset` needs to exist within DFC.

2.3 Design Constraints

As a base requirement, the accepted implementation for `SortDataset` in DFC must be functionally correct, meaning the output of DFC's solution must be identical to the output of the single-client solution. Additionally, it should achieve better performance than the single-client processing. The most straightforward way to measure this performance difference is to compare the processing time required for DFC's solution to that of the single-client solution.

Since DFC is a distributed system, an important design constraint is that no major processing should be done centrally. To achieve maximum parallelism, each step of the processing should utilize the entire cluster. As such, system resource allocation should be fair and uniform across all nodes. By having all targets work independently and in parallel on smaller chunks of the entire workload, modularity is achieved, and important features such as checkpointing can be implemented. Specifically, if K out of N targets fail during this processing, only K/N of the total workload should be recomputed.

Finally, the solution needs to implement an abstract framework which can be easily extended to support different compressed and uncompressed archive file types (e.g. .tar, .tgz, .msgpack), as well as different sorting schemes (e.g. alphanumerically by file name, by file content, reverse alphanumerically).

3 Accepted Solution

3.1 Description

The accepted solution is a distributed dataset sorting framework within DFC composed of three major phases: file-to-metadata extraction, intra-cluster sorting of metadata, and metadata-to-file creation. This `SortDataset` implementation is triggered via an HTTP endpoint on the primary proxy. As a part of the request to this endpoint, a client provides a request body specifying the details of the operation: information to identify the dataset (e.g. file name prefix, file number range), an enumerated type to specify the sorting heuristic to use, the desired resulting size for each new archive file created, the file type to operate on, and other adjustable options. Once the primary proxy receives this request, it forwards it to all targets in DFC in parallel so they can begin the first phase in the process. For developer ease when adding support for other archive file types and other sorting schemes, an abstraction was created around the three-phased framework. This abstraction uses Golang interfaces, which allow for the implementation of the abstraction to be determined at runtime based on dynamic values, or in other words, the request from the client [6].

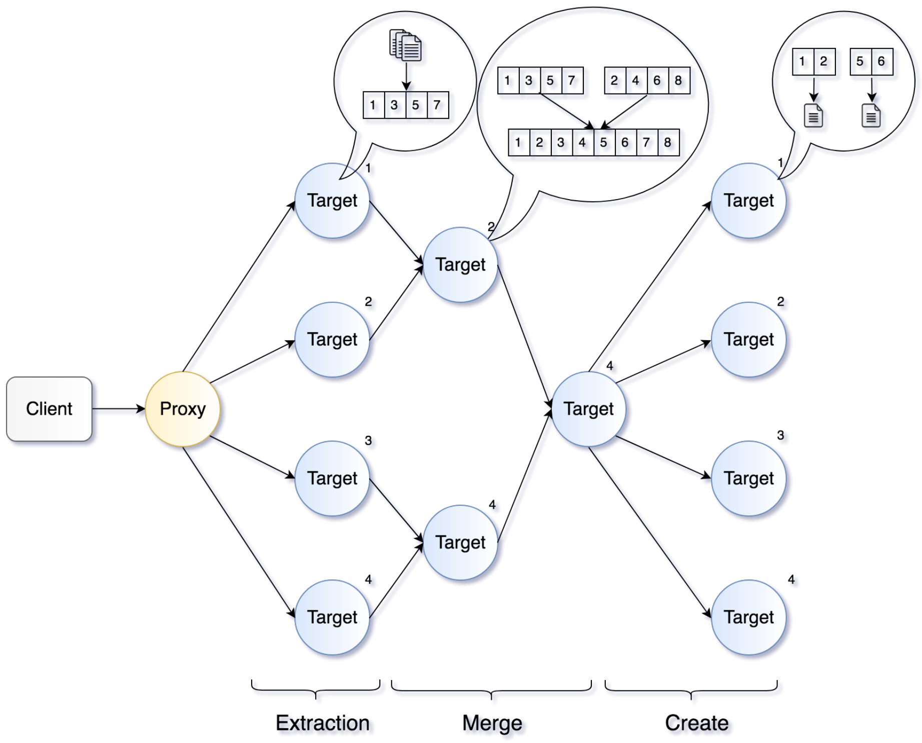


Figure 3: A high level overview of the three-phased solution.

Once a target receives the forwarded request from the primary proxy, it will (in parallel with all other targets), call an extraction function `E(s)` on each locally stored shard s belonging to the specified dataset. `E(s)` has two tasks: record content extraction and record metadata extraction. The record content extraction portion of `E(s)` attempts to store the contents of each record within a shard to a new *scatter-gather list*: a read-write data structure which is performance-optimized using non-contiguous blocks of memory. If a high watermark for local memory has been reached, the record contents are instead written to disk as a *workfile*: a temporary file for processing. As for the metadata extraction portion, a struct of metadata from each record in the shard including file name, file permissions, and file extended attributes is extracted. Once an array of metadata structs is produced from a shard, the array is sorted into a larger array stored in memory which constitutes the sorted, aggregated array of metadata for all shards belonging to the dataset which were found on the current target. Note that maximum parallelism is achieved by making every call (up to a certain client-defined limit) to `E(s)` in a *goroutine*: a Golang construct which represents a lightweight thread of execution [7]. Furthermore, input/output performance is optimized by treating memory as a cache for record contents and only writing to disk when reaching a high memory usage watermark.

As soon as a target finishes its part in the extraction phase, it moves on to the second phase: intra-cluster metadata sorting. In this phase, each target autonomously determines an arrangement involving itself and all other targets in the cluster using a salt that is provided by the primary proxy. By having the primary proxy provide the same salt to all targets in the cluster, every target's computed arrangement will be the same. Note that the salt is dynamically generated on the primary proxy, thus the arrangement of targets will not be the same for each `SortDataset` request. Based on this arrangement, targets in the cluster will begin participating in an ordered process of sending other targets their own metadata array in order to generate the complete, sorted array of metadata representing the new dataset. The algorithm used to sort all metadata arrays together across the cluster is based on the following pseudocode:

```
IntraClusterSort(arrangement):
        while length(arrangement) > 1:
                for target in arrangement:
                        if target != current target:
                                continue
                        if target is odd-indexed in arrangement:
                                send metadata array to be sorted into
                                metadata array of next target

                                return
                        else:
                                wait until metadata array of previous
                                target is sorted into metadata array
                                of current target

                                break
                remove each odd-indexed target from arrangement
```

Let s and r each represent a metadata array of a sending and receiving target, respectively. Since s and r are sorted, sorting the two together can be done in linear time by making $O(length(s) + length(r))$ comparisons, starting at the beginning of both arrays. By halving arrangement with each iteration of the outermost loop, this algorithm terminates in at most $log_2(length(arrangement)) = log_2(\#targets)$ iterations. By construction of the algorithm, the final target in arrangement will have the complete, sorted array of metadata in memory. At this point, the processing moves on to the third and final phase.

Once the final target has the complete, sorted array of metadata locally in memory, it will split the array into blocks of metadata – one corresponding to the contents of each new shard in the new dataset. The size of these blocks is based on the expected shard sizes for the new dataset. These blocks are then batched together into groupings on a per-target basis, with one grouping per target. The mapping between all groupings and targets is determined by choosing the target that will minimize intra-cluster network pressure during the shard creation phase. A heuristic which takes into consideration record content locality and the correct HRW-based location for each new shard quickly computes this mapping. Once this mapping is determined, all batches are sent out to their respective target in parallel. When a target receives its batch of metadata blocks, it calls a shard creation function `C(b)` in a goroutine for each block of metadata b. The function `C(b)` creates a new shard on disk based on b and has two main tasks: record content population, and record metadata updating. At a high level, the pseudocode for `C(b)` looks like the following:

```
C(b):
create an archive file f of the appropriate type on disk for b
for each metadata struct m in b:
        get the record contents corresponding to m from the target
        which produced m by the function E(s)

        write the record contents to f

        update the metadata for f for the record contents just written
        based on m
close f
```

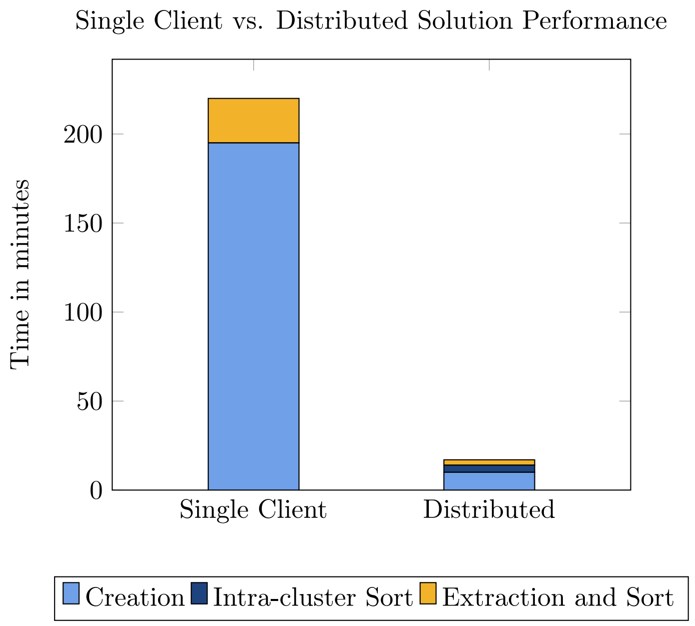
As soon as a target completes its work in the creation phase, it reports back to the primary proxy with the results. Once the primary proxy receives results from every target in the cluster, it reports the results back to the client, and `SortDataset` is complete.

3.2 Analysis of Accepted Solution

Maximum parallelism is achieved in this solution by doing the extraction and creation phases on a per-shard basis, with operations on each shard done in parallel along with all other shards. Furthermore, a balanced object distribution across DFC (by HRW) allows for yet another level of parallelism since each of the N targets in a cluster performs $1/N$ of the work required in parallel with all other targets.

In regards to the intra-cluster sorting phase, a major performance optimization is achieved by performing the sorting on small (less than 1KB) structs of metadata which represent the files, instead of the files themselves. Moreover, by sorting the arrays of metadata locally on each target before sorting again across targets, the cost of sorting across targets is optimized to only grow linearly with the sum of the sizes of the two sorted arrays.

In theory, the parallelism achieved in the extraction and creation phases and the optimizations achieved in the intra-cluster sorting phase should result in large reductions in processing time required. To confirm this, experiments comparing the performance of the distributed solution to the performance of a single client were conducted. To simulate this single client, I chose an arbitrary target in a DFC cluster to process the same workload that the distributed solution operated on. The same core functionality was used for extraction and creation – the only difference was that the single target did all file processing and sorting locally. For hardware, a cluster of 10 Amazon EC2 h1.16xlarge instances with 12 x 100GB HDDs (100MB/s) was used. The dataset used was 1TB in size, composed of 100,000 uncompressed tarballs. Each tarball contained 10 randomly named files of size 1MB. When performing the operation to sort alphanumerically by file name and generate new shards of size 10MB each, the results were as follows:



As illustrated in the graph above, the distributed solution completed the processing over 12 times faster than the single client. Note that the data for the single client processing time shows that the majority of the time is spent during the shard extraction phase. These results align with earlier predictions of system resources being a bottleneck since network, memory, and disk utilization on the single client will all be extremely high when ingesting and extracting the 1TB dataset locally.

4 Conclusions

By taking advantage of the distributed nature of DFC to achieve a high degree of processing parallelism and performing the intra-cluster sorting of metadata intelligently, a highly performant solution to the problem was implemented. The solution is comprised of three heavily parallelized phases: file-to-metadata extraction, intra-cluster sorting of metadata, and metadata-to-file creation. With this implementation, a 12X performance improvement was seen in an experiment with 10 DFC targets and a dataset of size 1TB.

Since DFC is an object storage solution tailored to AI applications, this dataset sorting-and-resharding feature will be a great asset to researchers who are looking for performance improvements when computing AI training and inference workloads. Furthermore, by adding this service to DFC, the utility of this software product as a solution for AI applications increases. Although it is yet to be measured, the impact of this framework in the broader worlds of academia and industry alike may be seen in years to come over the continual growth of not only this `SortDataset` implementation, but DFC as a whole.

Acknowledgements

I would like to thank Alex Aizman for his mentorship and and diligent guidance throughout the design and development of this project. Furthermore, I would like to thank NVIDIA for providing the appropriate tools and resources to effectively work on this problem. Finally, I would like to thank the staff of the department of Software Engineering at the University of Waterloo for providing pedagogical resources that were useful to me throughout the writing of this text.

References

[1] "About NVIDIA Corporation," *NVIDIA*. [Online]. Available: https://www.nvidia.com/en-us/about-nvidia/. [Accessed: 09-Sep-2018].

[2] NVIDIA, "NVIDIA/dfcpub," *GitHub*. [Online]. Available: https://github.com/NVIDIA/dfcpub. [Accessed: 09-Sep-2018].

[3] NVIDIA, "NVIDIA/dfcpub," *GitHub*. [Online]. Available: https://github.com/NVIDIA/dfcpub#cache-rebalancing. [Accessed: 11-Sep-2018].

[4] A. Liang and T. M. Breuel, 15-Jul-2018.

[5] A. Liang and A. Aizman, 10-Aug-2018.

[6] "Interfaces," *A Tour of Go*. [Online]. Available: https://tour.golang.org/methods/9. [Accessed: 10-Sep-2018].

[7] "Goroutines," A Tour of Go. [Online]. Available: https://tour.golang.org/concurrency/1. [Accessed: 10-Sep-2018].