Storage Benchmarking with Deep Learning Workloads

Peng Cheng
University of Chicago
Chicago, IL
pcpeng@uchicago.edu

Haryadi S. Gunawi
University of Chicago
Chicago, IL
haryadi@cs.uchicago.edu

ABSTRACT

In this AI-driven era, while compute infrastructure is often the focus, storage is equally important. Loading many batches of sample randomly is the common workload for deep learning (DL). Such iterative small random reads impose nontrivial I/O pressure on storage systems. Therefore, we are interested in exploring the optimal storage system and data format for storing DL data and the possible trade-offs. In the meantime, object storage is usually preferred because of its scalability characteristics, rich metadata, competitive cost, and its ability to store unstructured data.

This motivates us to benchmark two object storage systems: MinIO, Ceph. As a comparison, we also benchmark three popular key-value storage databases: MongoDB, Redis, and Cassandra. We explore the impact of different parameters including storage location, access pattern, storage disaggregation granularity, and data format. For each parameter, we summarize the benchmark results and give some suggestions. Overall, although the optimal storage system can be highly workload-specific, we think our benchmarks provide some insights about how to reach it.

KEYWORDS

Storage Systems, Object Storage, Database, Deep Learning, Performance Evaluation

ACM Reference Format:
Peng Cheng and Haryadi S. Gunawi. 2020. Storage Benchmarking with Deep Learning Workloads. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Deep learning (DL) has become a key technique for solving complex problems in scientific research and discovery. It is substantially challenging because it has to deal with massive quantities of multi-dimensional data. A distributed storage system formed by networking together with a large number of storage devices provides a promising mechanism to store the massive amount of data with high reliability and availability. Deep Learning system architects strive to design a balanced system where the computational accelerator – FPGA, GPU, etc, is not starved for data. Feeding training data fast enough to effectively keep the accelerator utilization high is difficult when utilizing dedicated hardware like GPUs. In addition, as accelerators are getting faster, the storage media & data buses feeding the data have not kept pace and the ever increasing size of training data further compounds the problem.

The random file access pattern in DL training is mainly required by the use of stochastic gradient descent (SGD) [34] model optimizer. The SGD model optimizer requires loading a mini-batch of training samples in a randomized order for every iteration. This is important for accelerating the model's convergence speed and decreasing the noise learned from the input sequence. However, such iterative loading of samples imposes nontrivial I/O pressure on storage systems which are typically designed and optimized for large sequential I/O [27, 35].

Potential bottlenecks to store large collections of small files have been pointed out across the file system community [5, 33], but none of these studies specifically considered the problem in the context of deep learning. In addition, to avoid overfitting caused by predefined input patterns, each batch of sample files must be selected randomly. This presents another challenge to the conventional I/O services that are highly optimized for large sequential I/O. To exploit the best performance, deep learning frameworks (e.g., TensorFlow [1], Caffe [19]) contain their own input methods. For example, TensorFlow introduces Dataset API for importing dataset from different formats. However, their data reading performance depends on underlying conventional file systems (e.g. standard POSIX file system) that are designed for general usage instead of being optimized for large-scale training.

In this report, we explore using object storage systems and key-value databases as solutions for storing deep learning datasets, and benchmark these systems with different parameters, including storage location, access pattern, storage disaggregation granularity, and data format. The results suggest that although the optimal solution may be workload-specific, proper configurations in various parameters can improve storage system performance for deep learning workloads.

In the remainder of this report we cover the background in Section 2 and review related work in Section 3. In Section 4, we introduce our methods. Finally, we present our results in Section 5, discuss some future work in Section 6, and summarize our work in Section 7.

2 BACKGROUND

In this section, we review the background of the storage disaggregation, I/O in deep learning, and object storage.

2.1 Storage Disaggregation

2.1.1 Resource Disaggregation. In cloud computing, the periods of peak demands can be short-lived, and this can result in significant

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

resource underutilization [4] because it is difficult to determine an exact amount of resources on every server to meet different applications' unique behaviors. Similar problems have been observed in large-scale data centers hosted by HPE [16], Intel [18], and Facebook [38]. Thus an emerging paradigm for resource provisioning called resource disaggregation is quickly gaining popularity. Many efforts have been carried out recently on resource disaggregation in both academia [2, 11] and industry [10, 17]. The goal of disaggregation is to decouple different resources or disaggregate a large amount of a single resource so that fine-grained allocations can be made to meet the dynamic demands of applications at run-time.

2.1.2 *Decoupled Storage and Compute.*

With resource aggregation, the resources are physically separated from the compute servers and accessed remotely, which allows each resource technology to evolve independently and supports increased configurability of resources. In the context of HPC, several projects have provided mechanisms for disaggregating GPU accelerators [29, 31]. Nowadays, decoupled storage and compute is becoming the mainstream considering the benefits of scalability, availability, and cost.

2.1.3 *Storage Disaggregation for Deep Learning.*

The provisioning of storage resources can be particularly hard because the actual demand may be a complex combination of required capacity, throughput in terms of I/O operations or files per second, bandwidth (bytes per second), latency, etc. For example, [30] claimed there is only 40% or less of the raw bandwidth of the flash memory in the SSDs is delivered by the storage system to the applications. Disaggregating storage resources can help amortize the total system building cost and maximize the performance-per-dollar of computing infrastructure. There have been storage disaggregation solutions to improve storage resource utilization on large-scale data centers [22, 28, 30]. These proposed solutions aim to solve the problem at the system level. [42] presents a user-level, read-optimized file system on top of non-volatile memory (NVM) devices for deep learning applications called DLFS, but their work focuses on in-memory sample directory for fast metadata management. To the best of our knowledge, there has been no effort to leverage storage disaggregation for deep learning by exploring storing each training sample with its corresponding label as an individual object.

2.2 I/O in Deep Learning

Gradient descent is one of the most popular algorithms to optimize parameters of the DNNs. Mini-batch gradient descent, often referred to as SGD [34], is commonly used because of its fast training speed and low memory requirements. However, there are a number of challenges for the effective utilization of storage resources on large-scale HPC systems.

2.2.1 *Increased mini-batch size.*

The popularity of SGD has led to many recent optimizations to leverage increasing compute power. For example, it is important to batch more input samples, i.e., increasing the mini-batch size, in each training iteration. Many studies [12, 15, 36, 39] have reported that they can finish the training of ResNet- 50 [15] with ImageNet [9] at fast speeds. In these studies, the batch size has increased from 256 [15] to 80K [39] over the past few years. Compared to the conventional demands on high bandwidth and low latency, the increased mini-batch size requires much higher throughput so that more samples can be trained in each iteration.

2.2.2 *Many small random samples.*

In deep learning training, a large number of training samples are expected to arrive in a random order to speed up the convergence speed and reduce the training biases caused by fixed input sequences.

Many popular datasets contain small samples. For example, the ImageNet dataset consists of many small samples (every raw image file is an individual sample), about 75% of samples are less than 147 KB [42]. A similar trend is also shown in [7].

Working with these datasets result in many random reads to the storage system. This is a challenging I/O pattern because it cannot benefit from the traditional storage systems that are typically designed for large sequential I/O patterns.

Although we can preprocess small samples into large batched files (e.g., TFRecord format) to avoid small random I/O, the existing sample shuffling method cannot support global data shuffling, and the size of shuffle buffer limits the shuffling result.

For example, when using TFRecord files in TensorFlow, every TFRecord file is sequentially read in a fixed size shuffle buffer. However, if the size of the shuffle buffer is not large enough, the learner only obtains partially shuffled samples, which reduces the training accuracy.

2.3 Object Storage

2.3.1 *Motivation.*

We can think of object storage as a hybrid storage of file systems and block storage. On the one hand, many applications nowadays depend less on the traditional POSIX file systems. For example, applications are satisfied with storing images in a file system that only supports a flat namespace and weak consistency. Also, content sharing applications can also painlessly devise their own ways to handle concurrent modifications without resorting to file system locks. On the other hand, although a reduction of file system features frequently fuels scalability, developers still need storage services to free them from the burden of managing blocks and various other storage metadata. To this end, object storage systems are often implemented with eclectic designs: they are less sophisticated than file systems but still more intelligent than block devices. The rise of cloud computing has also assisted the emergence of object storage. Object storage echoes the very spirit of cloud computing. It provides people with an unlimited and on-demand data depository accessible from anywhere in the world, and it helps to achieve data consolidation and economy of scale, both facilitating cost-effectiveness. With cloud computing getting increasingly adopted, so are objects [41].

2.3.2 *Popularity of Object storage.*

With object storage systems being increasingly recognized as a preferred way to expose one's storage infrastructure to the web, the past few years have witnessed an explosion in the acceptance of these systems

With our world being increasingly connected and enriched by newly emerged computing technologies, the data produced is gigantic [40]. To combat such data tsunami, object storage has been introduced to fulfill the common need of a scale-out storage infrastructure, with acceptance of these systems growing every year [3].

2.3.3 Storage Interface. Object storage systems are designed as RESTful web services and are accessed via HTTP requests. To use object storage systems, people create containers and put objects into these containers for storage. Objects are just like files in regular file systems, though they cannot be locked or updated partially. Containers are identical to directories except that they cannot be nested. Both objects and containers are identified by unique names distinguishing one from each other.

2.3.4 Properties of Object Storage. In object storage, the data is arranged in discrete units called objects and is kept in a single repository. Object storage volumes work as modular units: each is a self-contained repository that owns the data, a unique identifier that allows the object to be found over a distributed system, and the metadata that describes the data [37]. In general, object storage is optimal to store DL data because of its scalability characteristics, rich metadata, and its ability to store unstructured data.

The ability to store unstructured data. AI and ML learn from many different data types, which requires varying performance capabilities. Therefore, storage systems must include the right mix of storage technologies – a hybrid architecture – to meet the simultaneous needs for scale and performance. The ability to store different types of unstructured data makes object storage a great fit to storage DL data.

Rich Metadata. File systems place some structure onto data, putting file objects into hierarchies (folders/directories) and attaching metadata to those objects. However, the metadata is typically only based on the information needed to store the file (time created, time updated, access rules). Object stores go one step further and remove the folder hierarchy. Objects are stored with extensible metadata that is typically highly searchable

High Scalability. Object storage is ideal for storing content that can grow without bounds. Use cases include backups, archives and static web content such as images. For storing a very large amount of data across different locations with extensive metadata, object storage is ideal.

Some of the popular object storage systems are MinIO, Ceph, and AIstore.

3 RELATED WORK

The most closely related work includes performance analysis of deep learning applications' workflow. A few recent studies [13, 14, 20, 25] have documented the evaluation of different DL applications on different HPC systems, but all of these mainly deal with the computation characterization. [24] evaluated how various hardware configurations that affect the overall performance of deep learning, including storage devices, main memory, CPU, and GPU. They compared training time using four storage devices: single disk, single SSD, Disk Array, SSD Array. Their work only focus on the effect of various hardware configurations, but our work also explored the impact of different access patterns as well as data formats.

There are also some efforts made on I/O profiling and optimization for DL training workloads [7, 24, 26, 32]. For example, [26] evaluate image storage systems for training deep neural networks, including key-value databases, file systems, and in-memory Python/C++ array. [24] evaluate how various hardware configurations (e.g. storage devices, main memory, CPU, and GPU) affect the overall performance of deep learning. Our work focuses on the general-purpose storage systems, especially involving object storage systems, and aims to characterize the impact of different storage parameters.

4 METHODS

4.1 Experiment Setup

We benchmark 6 storage systems, including two object storage systems (MinIO, Ceph) and three key-value databases (MongoDB, Redis, Cassandra), with the ext4[1] file system. The testbed is built with d430 physical nodes at Emulab[2], which are connected by 1Gbps links and form a local area network (LAN). The datasets used for benchmarking are MNIST and CIFAR-10 (Table 2), two simple but popular datasets for Computer Vision tasks. We assume these datasets are stored in exclusive storage systems and downloaded to a client for further machine learning use.

Table 1: d430 Node Details

Component	Description
CPU	2 × Intel Xeon E5-2630v3 (8C16T), 2.4GHz
Memory	64GB DDR4 2133
NIC	Intel i350 GbE NIC
Storage	[l]200 GB 6Gbps SATA SSD,
2 × 1 TB 7200 RPM 6 Gbps SATA disks	

Table 2: Dataset Details

Dataset	Size (Local, MB)	Description
MNIST	236	60,000 images of hand-written digits
CIFAR-10	952	60,000 32×32 images in 10 classes

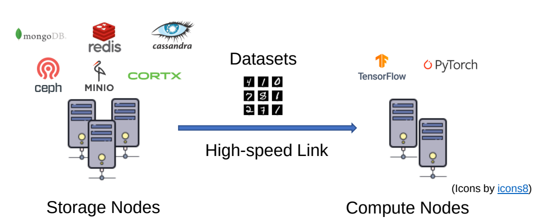


Figure 1: Overview of the Benchmark Approach

4.2 Metrics

4.2.1 Download Time. While large deep learning models with enormous parameters are emerging (e.g. GPT-3 [6]), the cost of training is increasing rapidly as well. Therefore, the end-to-end training time has become an important metric for evaluating a model [8].

[1] In some cases (e.g. Ceph) it can be XFS, we will check this and re-run some experiments if necessary after emulab nodes are available again.
[2] https://www.emulab.net/portal/frontpage.php

Recent analysis shows that I/O can take up 90% of the total training time [7], and if we load datasets from remote storage nodes, we believe "download time" is a prominent part of the I/O cost.

In the following experiments, download time is defined as the time from the first request for training data to the last one, which consists of the data processing time in storage systems and network transfer time.

4.2.2 Disk Usage. Given the limited storage resources, in order to store more data, we would like to reduce the disk usage of datasets. Serialization techniques (e.g. Python's pickle, blob) that convert objects into byte streams can partly achieve that, while compression (e.g. LZ4, Snappy) directly aims at that and has been widely used in storage systems. We check the disk usage of uploaded data using the APIs of those storage systems.

4.3 Benchmark Parameters

4.3.1 Location. At storage nodes, the data can be stored in memory, solid-state drive (SSD) or disk—in this order both the cost and speed decrease. Considering the huge difference in the cost per GB, we are interested in the impact of storage location on the download time.

4.3.2 Access Pattern. Gradient descent, which is an important method for deep learning model training, works on one or a batch of samples in each iteration. The access pattern includes two parameters: (1) number of objects per request, which is related to the batch size in machine learning; (2) random/sequential access, which depends on whether we fetch the data randomly or in order. We expect the access pattern can influence the download time significantly.

4.3.3 Storage Disaggregation Granularity. We can choose to store single sample as an object, or combine multiple samples into an object. While in the former case we can have more fine-grained metadata and fetch batch of samples in any size from the remote system, in the latter case fewer requests are needed if we fetch same amount of data—so there exists a trade-off.

4.3.4 Data Format. We can simply store the data in raw formats (e.g. images, text). However, data serialization is also a common method for storage, transfer, and distribution purposes. Data format can influence the download time and disk usage directly by changing data size or indirectly through compression ratio. In Cassandra experiments we compare the download time and disk usage with three data formats: raw files, Python's pickle, and Binary Large Object (blob)

4.4 Benchmark Implementation

We parse the raw files of dataset and stored one/multiple image(s) with its/their label(s) as a single object using Python3's pickle serialization, and then upload them or the raw files to each storage system. To benchmark the storage systems on SSD or disk, we manually set data directories at mounted SSD or disk and use the vmtouch[3] or restart the systems to clean the cache. Fetching data from memory is a little tricky—Redis is memory-based, for the other systems we pre-load the data into memory by querying once and then begin the tests.

At client side, we load the data using Python drivers corresponding to the storage systems. We only simulates the process of loading training data first, and then incorporated real model training. For random read in batch, we shuffle the indices using a fixed random seed with Numpy, slice them into batches, and request the data from storage systems using the batches of indices, which corresponds to the mini-batch gradient descent; for sequential read, we fetch the objects one by one or the full dataset at one time. It is noted that while databases provide operations for querying multiple samples at a time (Table 3), object storage systems don't. Therefore, we also consider disaggregating the dataset at a coarser granularity (more samples per object). Each performance test is repeated 3 times and the average value is reported.

Table 3: Query Operations for Multiple Objects at a Time

System	Operation	Example
MongoDB	in	collection.find({"_id": {"$in": idx_batch}})
Cassandra	WHERE...IN	SELECT * FROM MNIST WHERE id IN ?
Redis	hmget	hmget("mnist", filenameList)

5 RESULTS

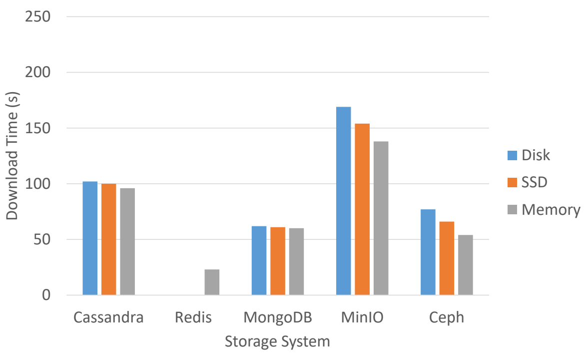
5.1 System and Location

We first analyze the impact of system choices and storage locations, downloading pickled datasets from different systems with sequential read and single sample per request. Figure 2(a) and 2(b) show the download time of MNIST and CIFAR-10 respectively. Note that the size of CIFAR-10 is about 4 times larger than MNIST, but the download time only increases as much as 30% (Ceph, disk), indicating some processing time that's not linear in the dataset size dominates the download time.

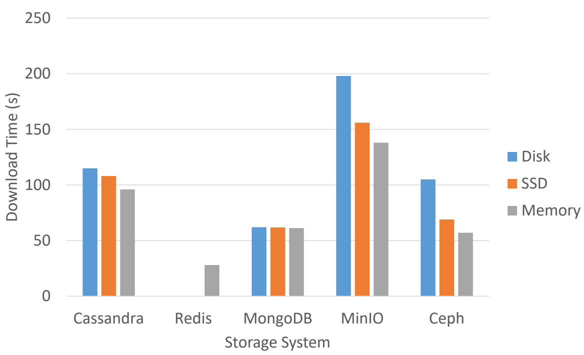
Bars in each group present the download time from different storage locations. It's not surprising that memory is faster than SSD, and SSD is faster than disk. The differences reflect that object storage systems (MinIO, Ceph) are more sensitive to the storage locations than the key-value databases (Cassandra, Redis, MongoDB), and they increase as dataset size increases. With Redis the data is only retrieved from memory, which makes it at least twice faster than the other storage systems. However, as memory is usually more limited than SSD and disk space on servers, Redis may not work with larger datasets.

As is mentioned, there exists processing time that is not linear in the dataset size, so we drill down to the Cassandra experiment, trying to break down the MNIST download time from Cassandra into reading data time, network transfer time, and processing data time (Figure 3). Running a client program on the server node is around 10% faster than running it on the client node, indicating network transfer time takes up approximately 10% of download time. It takes only 4.2 seconds (SSD) or 6.6 seconds (disk) to load (sequential read, 1 Sample per Request) 60000 files on the local file system, which means reading data time takes up approximately 5% of download time. This indicates Cassandra data processing takes up most of the time (~85%). So far this is still a black box to us,

[3] https://github.com/hoytech/vmtouch



(a) MNIST



(b) CIFAR-10

Figure 2: Download Time of Pickled Datasets by Storage System and Location (Sequential Read, 1 Sample per Request). Note that data is retrieved only from memory in Redis.

but we think it also means opportunities—there exists room for improvement by adjusting some system configurations (e.g. index, compression algorithms).

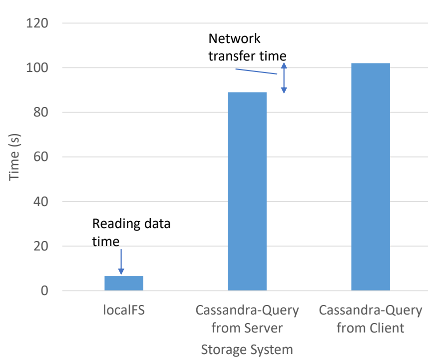


Figure 3: Cassandra Download Time=Reading Data Time + Network Transfer Time + Processing Data Time (Pickled MNIST, Disk)

Summary: As is expected, in terms of download time, memory<SSD<memory, but the impact of system choice is more significant. In addition, with high-speed network link, data processing time dominates the download time. This suggests we should choose a storage system with proper configurations according to the workload to get the best performance.

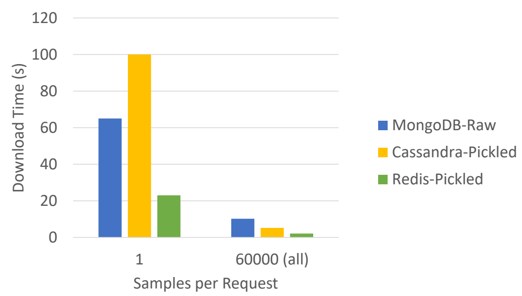
5.2 Access Pattern

Next we consider how the download time changes with the access pattern. Figure 4 presents the download time from three databases, which support query for one or multiple objects at a time, with different access patterns. In this experiment number of samples per object is fixed at 1. Figure 4(a) compares downloading samples one by one in order and downloading the full dataset. While downloading the full set is 6–20 times faster than downloading one sample at a time, this may not work when the full dataset is large (e.g. Youtube-8M). Therefore, we turn to requesting multiple samples randomly at a time in Figure 4(b), which corresponds to the model training process with stochastic gradient descent. The leftmost points also present the results of querying one sample at a time, but in a random order. The random access sees about 10% increase in download time, reflecting the penalty of random read compared with sequential read. A good news is that as we increase the number of samples per request, the download time decreases rapidly, from 2x faster at 4 to 10x faster at 256. Since the total network transfer time should be the same, we infer that the benefit comes from decreased query numbers and corresponding processing time. After the number of samples per request reaches 16, the decrease slows down, suggesting an area where we can find a number of samples per request that achieves a balance between the size limit and download time.

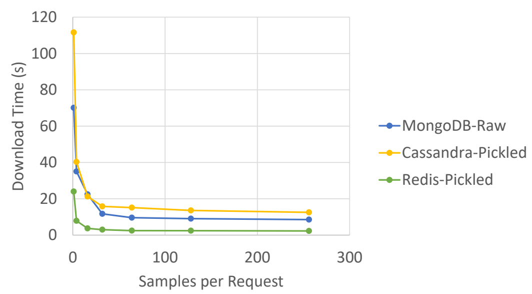
Summary: Random access sees some penalty, but increasing the number of samples per request using built-in operations can mitigate that and even match the speed of downloading the full dataset at one time, which is good for model training on batches of samples.

5.3 Storage Disaggregation Granularity

5.3.1 Key-value Databases. As key-value databases support query for multiple objects at a time, we then want to explore the impact of storage disaggregation granularity (i.e. number of samples per object). To compare cases like retrieving 4 one-sample objects and 1 four-sample object, we use "images per request" as the x-axis in Figure 5. As Figure 5 shows, in MongoDB, coarser-grained object can reduce < 10% download time when the number of images/request is small or < 1s when the number of images/request is large. In this case, changing the storage disaggregation granularity only affects the number of objects per request, and it seems that the download time is not so sensitive to this, so more fine-grained disaggregation can be favorable because of fine-grained management without much overhead.



(a) Sequential



(b) Random

Figure 4: Download Time of Pickled MNIST by Access Pattern.

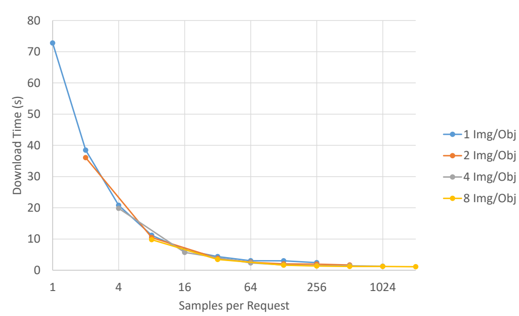


Figure 5: MongoDB Download Time by Storage Disaggregation Granularity and Number of Samples per Request (Pickled MNIST, Disk)

5.3.2 Object Storage. The object storage systems we select only support query for one object at a time, so we can only change the number of samples per request by changing the number of samples per object. Figure 6 shows the download time of MNIST and CIFAR-10 from MinIO (disk) by the number of samples per object, while downloading all samples in an object takes 0.95s (MNIST) and 2.97s (CIFAR-10) respectively. Note that the x-axis is log-2, so we nearly halve the download time by doubling the samples per object. This is more significant with MNIST, the smaller dataset, suggesting some query overhead that is linear to the number of requests may take up most of the time.

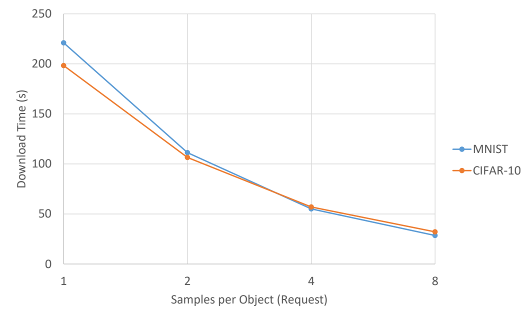
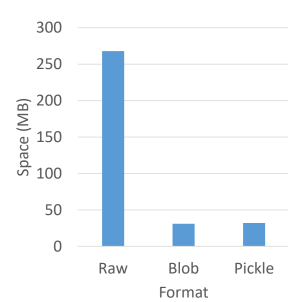


Figure 6: MinIO Download Time by Storage Disaggregation Granularity and Dataset (Disk)

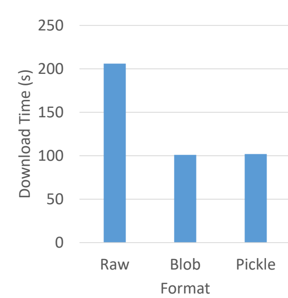
Summary: Experiments show that storage disaggregation granularity can affect download time, especially for object storage systems. The balance between performance and fine-grained management may depend on the workload (dataset).

5.4 Data Format

Figure 7 shows the disk usage and download time with different data formats of MNIST in Cassandra. Both blob and pickle only take up about 11% of the disk space used by raw images, and about 2/3 of the memory space when they are loaded into memory. The reason of the difference is that data serialization improves the compression ratio. In addition to the disk usage, data format also influences the download time (Figure 7(b)). Downloading blob or pickled MNIST is about 2x faster than downloading the raw files. Although the benefits of serialization come at a cost of local deserialization, the cost is much less than the benefits (for MNIST, only about 5 seconds with Python3's pickle. Cassandra's Python driver decodes blob automatically with minor cost).



(a) Disk Usage



(b) Download Time (Sequential, 1 Sample per Request)

Figure 7: Disk Usage and Download Time by Data Format (Cassandra, Pickled MNIST, Disk)

Summary: Data serialization reduces not only the disk usage of datasets but also the download time. Since the cost of deserialization is much less than the benefits, it can be a good idea to pre-process the raw datasets into suitable formats or use other techniques such as compression. For small images we think blob is a good choice.

5.5 Integration with Model Training

So far our experiments only simulate the process of loading training data from remote servers, which haven't incorporated actual model training. It will be interesting to see what will happen if we combine remote data loading and model training. Therefore, we implement training LeNet-5 [21] on MNIST with PyTorch [23] and CPU, and try loading data locally or from Redis [4], which performs the best in previous experiments but may not work for larger datasets that cannot fit in memory.

We define a *RedisMnist* class, override the *__getitem__* method with Redis Python driver's *hget* function that retrieves one sample at a time, and train the model with different batch sizes and numbers of workers, which enable multi-process data loading. Figure 8 shows the end-to-end training time from loading the data to the end of training. When the batch size is small (1 and 4), the training time is long and hide the loading data time, so results with local storage and Redis are similar. However, as batch size increases, I/O gradually becomes the bottleneck. In this case, loading from Redis takes 1.2–3.4-fold time compared with local storage. Increasing the loading workers significantly helps remote loading, but it still takes much longer to load the data remotely. MNIST is a relatively small dataset, which means the remote I/O bottleneck can be more severe for larger datasets—this shows the value of our work that aims to find the optimal remote storage configurations for deep learning workloads.

Summary: Current deep learning frameworks provide interfaces for loading data from remote sources, but efficient implementation (e.g. exploit database API that retrieves multiple objects at a time) needs further investigation. In addition, while long training time can hide the loading latency, loading data from remote sources becomes a bottleneck as training time decreases. Multi-process data loading can mitigate this but may be not enough.

6 FUTURE WORK

Better Integration with Model Training. If we only override the *__getitem__* method in Dataset class, the default behavior of DataLoader corresponds to concurrent random single reads, which is bad for performance. For key-value databases, query for multiple samples at a time is more efficient than multiple queries. To exploit this, we may need to carefully customize a data loader in PyTorch.

7 CONCLUSION

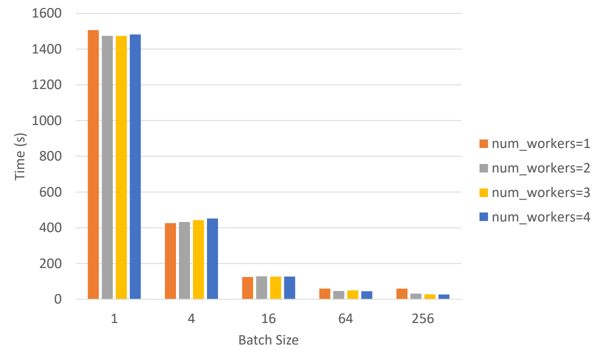
This AI-driven era requires high-performance storage systems as well as new compute infrastructure. We benchmark three key-value databases, and three object storage systems, which are potentially better solutions. Our results show that performance improvement can be achieved by choosing proper parameters including storage location, access pattern, storage disaggregation granularity, and data format.

ACKNOWLEDGMENTS

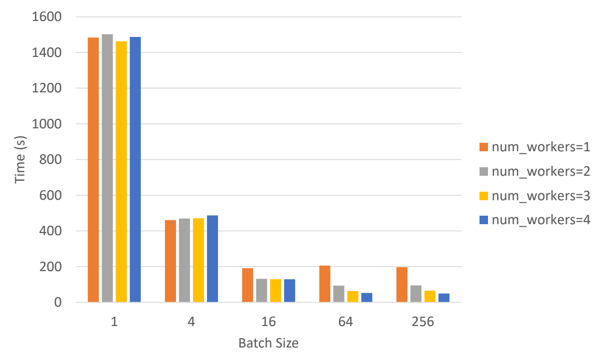
REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.
[2] Krste Asanović. 2014. Firebox: A hardware building block for 2020 warehouse-scale computers. (2014).
[3] Jeff Barr. 2012. Amazon S3 – 905 Billion Objects and 650,000 Requests/Second. https://aws.amazon.com/cn/blogs/aws/amazon-s3-905-billion-objects-and-650000-requestssecond/
[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture* 8, 3 (2013), 1–154.
[5] Doug Beaver, Sanjeev Kumar, Harry C Li, Jason Sobel, Peter Vajgel, et al. 2010. Finding a Needle in Haystack: Facebook's Photo Storage.. In *OSDI*, Vol. 10. 1–8.
[6] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[7] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. 2019. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th International Conference on Parallel Processing*. 1–10.
[8] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. *Training* 100, 101 (2017), 102.
[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
[10] Facebook. [n.d.]. Introducing Data Center Fabric, The Next-Generation Facebook Data Center Network. https://code:fb:com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-datacenter-network
[11] Peter X Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. 2016. Network requirements for resource disaggregation. In *12th {USENIX} Symposium on Operating

[4] https://github.com/ptrblck/pytorch_misc/blob/master/pytorch_redis.py



(a) Loading from Local Storage



(b) Loading from Redis

Figure 8: End-to-end Training Time by Data Locations, Number of Workers, and Batch Size (#Samples/Batch).

Systems Design and Implementation ({OSDI} 16). 249–264.

[12] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677* (2017).

[13] Jiazhen Gu, Huan Liu, Yangfan Zhou, and Xin Wang. 2017. Deepprof: Performance analysis for deep learning applications via mining gpu execution patterns. *arXiv preprint arXiv:1707.03750* (2017).

[14] Mauricio Guignard, Marcelo Schild, Carlos S Bederián, Nicolás Wolovick, and Augusto J Vega. 2018. Performance characterization of state-of-the-art deep learning workloads on an ibm" minsky" platform. In *Proceedings of the 51st Hawaii International Conference on System Sciences*.

[15] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[16] HP. [n.d.]. Moonshot System: The Worlds First Software-Defined Servers. http://h10032.www1.hp.com/ctg/Manual/c03728406.pdf/

[17] Intel. [n.d.]. Intel Rack Scale Design. https://www:intel:com/content/www/us/en/architecture-and-technology/rack-scale-design-overview:html

[18] Intel. [n.d.]. Intel RSA. https://www.intel.com/content/www/us/en/architectureand-technology/rack-scale-design-overview.html

[19] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[20] Yuriy Kochura, Sergii Stirenko, Oleg Alienin, Michail Novotarskiy, and Yuri Gordienko. 2017. Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes. In *Conference on computer science and information technologies*. Springer, 243–256.

[21] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.

[22] Sergey Legtchenko, Hugh Williams, Kaveh Razavi, Austin Donnelly, Richard Black, Andrew Douglas, Nathanaël Cheriere, Daniel Fryer, Kai Mast, Angela Demke Brown, et al. 2017. Understanding rack-scale disaggregated storage. In *9th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*.

[23] Eryk Lewinson. 2020. Implementing Yann LeCun's LeNet-5 in PyTorch. https://towardsdatascience.com/implementing-yann-lecuns-lenet-5-in-pytorch-5e05a0911320

[24] Jingjun Li, Chen Zhang, Qiang Cao, Chuanyi Qi, Jianzhong Huang, and Changsheng Xie. 2017. An experimental study on deep learning based on different hardware configurations. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*. IEEE, 1–6.

[25] Xiaqing Li, Guangyan Zhang, H Howie Huang, Zhufan Wang, and Weimin Zheng. 2016. Performance analysis of gpu-based convolutional neural networks. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 67–76.

[26] Seung-Hwan Lim, Steven R Young, and Robert M Patton. 2016. An analysis of image storage systems for scalable training of deep neural networks. *system* 5, 7 (2016), 11.

[27] Timothy Prickett Morgan. 2018. Removing The Storage Bottleneck for AI. www.nextplatform.com/2018/03/29/removing-the-storage-bottleneck-for-ai

[28] Mihir Nanavati, Jake Wires, and Andrew Warfield. 2017. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 17–33.

[29] Minoru Oikawa, Atsushi Kawai, Kentaro Nomura, Kenji Yasuoka, Kazuyuki Yoshikawa, and Tetsu Narumi. 2012. DS-CUDA: a middleware to use many GPUs in the cloud environment. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 1207–1214.

[30] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 471–484.

[31] Antonio J Peña, Carlos Reaño, Federico Silla, Rafael Mayo, Enrique S Quintana-Ortí, and José Duato. 2014. A complete and efficient CUDA-sharing solution for HPC clusters. *Parallel Comput.* 40, 10 (2014), 574–588.

[32] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. 2017. Towards scalable deep learning via I/O analysis and optimization. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 223–230.

[33] Kai Ren and Garth Gibson. 2013. TABLEFS: enhancing metadata efficiency in the local file system. In *Proceedings of the 2013 USENIX conference on Annual Technical Conference*. 145–156.

[34] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).

[35] Open Data Science. 2018. Data Storage Keeping Pace for AI and Deep Learning. https://medium.com/predict/data-storage-keeping-pace-for-ai-and-deep-learning-ad3e75e1c67a

[36] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489* (2017).

[37] Arun Taneja. 2019. Advantages of object storage and how it differs from alternatives. https://searchstorage.techtarget.com/feature/How-an-object-store-differs-from-file-and-block-storage

[38] Jason Taylor. 2015. Facebook's data center infrastructure: Open compute, disaggregated rack, and beyond. In *Optical Fiber Communication Conference*. Optical Society of America, W1D–5.

[39] Masafumi Yamazaki, Akihiko Kasagi, Akihiro Tabuchi, Takumi Honda, Masahiro Miwa, Naoto Fukumoto, Tsuguchika Tabaru, Atsushi Ike, and Kohta Nakashima. 2019. Yet another accelerated sgd: Resnet-50 training on imagenet in 74.7 seconds. *arXiv preprint arXiv:1903.12650* (2019).

[40] N Yezhkova, L Conner, R Villars, and B Woo. 2010. Worldwide enterprise storage systems 2010–2014 forecast: recovery, efficiency, and digitization shaping customer requirements for storage systems. *IDC, May* (2010).

[41] Qing Zheng, Haopeng Chen, Yaguang Wang, Jiangang Duan, and Zhiteng Huang. 2012. Cosbench: A benchmark tool for cloud object storage services. In *2012 IEEE Fifth International Conference on Cloud Computing*. IEEE, 998–999.

[42] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. 2019. Efficient User-Level Storage Disaggregation for Deep Learning. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 1–12.

A FULL RESULTS

Table 4: MNIST Download Time (s) by Storage System, Access Pattern, Data Format (Pickle if not specified), and Location. ("LocalFS" = local file system, "Seq"=sequential, "Full"=full set)

System	Access Pattern	Memory	SSD	Disk
LocalFS	Seq, 1	1.3	4.2	7
MinIO	Seq, 1	138	154	169
Ceph	Seq, 1	54	66	77
Redis	Seq, 1	23	N/A	N/A
	Seq, Full	2.1	N/A	N/A
MongoDB-raw	Seq, 1	58	62	65
	Seq, Full	7	9.1	10.2
Cassandra	Seq, 1	96	100	102
	Seq, Full	4.6	5.2	5.4
Cassandra-blob	Seq, 1	99	99	101
	Seq, Full	3	3.7	3.8
Cassandra-raw	Seq, 1	196	200	206
	Seq, Full	45	46	47

Table 5: MNIST Download Time (s) by Random Access

# of Samples per Request	1	4	16	32	64	128	256
MongoDB-Raw	70.12	35.05	22.44	11.71	9.6	9.09	8.56
Cassandra-Pickled	111.6	40.3	21.3	15.81	15.15	13.59	12.53
Redis-Pickled	24.02	7.87	3.72	3.01	2.46	2.44	2.28

Table 6: CIFAR-10 Download Time (s) by Storage System, Access Pattern, Data Format (Pickle if not specified), and Location.

System	Access Pattern	Memory	SSD	Disk
LocalFS	Seq, 1	1.4	7	19
MinIO	Seq, 1	138	156	198
Ceph	Seq, 1	57	69	105
Redis	Seq, 1	28	N/A	N/A
	Seq, Full	8	N/A	N/A
MongoDB-raw	Seq, 1	80	88	87
	Seq, Full	23	30	31
Cassandra	Seq, 1	96	108	115
	Seq, Full	20	20	21
Cassandra-blob	Seq, 1	97	104	113
	Seq, Full	12.5	14	15
Cassandra-raw	Seq, 1	330	350	375
	Seq, Full	132	142	139